Pametna zgrada kao višeagentni sustav

Hlevnjak, Tomislav

Master's thesis / Diplomski rad

2019

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:868696

Rights / Prava: Attribution-NonCommercial-ShareAlike 3.0 Unported/Imenovanje-Nekomercijalno-Dijeli pod istim uvjetima 3.0

Download date / Datum preuzimanja: **2024-04-23**

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

zir.nsk.hr

DIGITALNI AKADEMSKI ARHIVI I REPOZITORIJI

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Hlevnjak

Pametna zgrada kao višeagentni sustav

DIPLOMSKI RAD

Varaždin, 2019.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Tomislav Hlevnjak

Matični broj: 0246030770

Studij: Informacijsko i programsko inženjerstvo

Pametna zgrada kao višeagentni sustav

DIPLOMSKI RAD

Mentor:

Dr. sc. Igor Tomičić

Varaždin, rujan 2019.

Tomislav Hlevnjak

Izjava o izvornosti

Izjavljujem da je moj diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom radu bit će objašnjeni pojmovi agenta i višeagentnog sustava (engl. *Multi Agent System - MAS*) te komunikacija između agenata i način kako ih povezati u pametni, autonomni višeagentni sustav.

Zatim će biti predstavljen koncept pametne električne mreže (engl. *smart grid*), pametnog brojila (engl. *smart meter*) te upravljanje potrošnjom električne energije pomoću metode odgovora na potražnju (engl. *Demand Response - DR*).

Na kraju će biti obrađena tema pametne zgrade kao višeagentnog sustava, a fokus će biti na uštedi električne energije. Kako bi se uštedila električna energija zgrada će biti dio pametne električne mreže te će preko pametnog brojila sudjelovati u „*Demand Response*“ programu. Zgrada će imati i solarne panele te će preko vremenske prognoze predviđati proizvodnju energije, a pametno brojilo će odlučivati je li u pojedinom trenutku bolje koristiti električnu energiju koju proizvode i skladište solarni paneli ili kupovati struju od poslužitelja. Prikazat će se implementacija nekoliko potrošačkih agenata, pametnog brojila, agenta elektrane i solarnih panela implementiranih pomoću Java Agent DEvelopment Environment (nadalje u tekstu JADE) programskog okvira te će se analizirati potrošnja električne energije kroz dvije simulacije i napraviti usporedba s potrošnjom u običnoj zgradi.

Ključne riječi: agent; višeagentni sustav; demand response; pametna mreža; pametna zgrada; pametno brojilo; jade; java

Sadržaj

1. Uvod 1
2.1. Okolina 3
2.2.1. Apstraktna arhitektura 5
2.2.1.1. Čisto reaktivni agenti 6
2.2.1.2. Percepcija 6
2.2.1.3. Agenti sa stanjem 7
2.2.2. Konkretna arhitektura 8
2.2.2.1. Agenti bazirani na logici 8
2.2.2.2. Reaktivni agenti 9
2.2.2.3. BDI agenti 11
2.2.2.4. Slojeviti (hibridni) agenti 13
2.3. Program agenta 16
2.3.1. Jednostavni refleksivni agenti 17
2.3.2. Refleksivni agenti temeljeni na modelu 18
2.3.3. Agenti temeljni na ciljevima 19
2.3.4. Agenti temeljeni na korisnosti 20
2.3.5. Agenti sa sposobnošću učenja 21
3. Višeagentni sustavi 23
3.1. Karakteristike višeagentnog sustava 25
3.2. Usporedba sa sličnim sustavima 26
3.2.1. Usporedba s ekspertnim sustavima 26
3.2.2. Usporedba s objektima 27
4. Pametna mreža 28
4.1. Pametno brojilo 30
5. Odgovor na potražnju 32
6. Pametna zgrada 33
7. Implementacija 36
7.1. Korištene tehnologije 36
7.1.1. Java Agent DEvelopment Framework 36
7.1.1.1. Klasa Agent 38
7.1.1.2. Klasa Behaviour 40
7.1.1.3. Interakcijski protokoli 41
7.1.1.4. Komunikacija Agenta i ACLMessage klasa 42
7.2. Opis sustava 44
7.3. Ontologija 46
7.3.1. Klasa Demand 47
7.3.2. Klasa HouseHoldingDemand 47

7.3.3. Klasa SolarDemand 47
7.3.4. Klasa PriceSignal 48
7.3.5. Klasa Switch 48
7.3.6. Klasa Vocabulary 49
7.3.7. Klasa SmartBuildingOntology 49
7.4. Struktura agenata 50
7.4.1. Agent PowerPlantUtility 51
7.4.2. Agent SmartMeter 57
7.4.3. Agent HouseHoldingAppliance 68
7.4.4. Agent SolarPanel 73
7.5. Primjer izvođenja 76
7.5.1. Simulacija 1 76
7.5.2. Simulacija 2 79
7.6. Analiza rezultata 82
7.6.1. Simulacija 1 83
7.6.2. Simulacija 2 84
7.7. Usporedba s drugim sustavima 85
8. Zaključak 87

1. Uvod

Informacijske i komunikacijske tehnologije razvijaju se sve brže i brže, internet posjeduje gotovo svako kućanstvo, a sve više i više kućanskih aparata i uređaja koje koristimo u kućanstvu ima mogućnost spajanja na internet. Stoga ne čudi da se rodila ideja pametne zgrade u kojoj bi uređaji koristili te tehnologije i mogućnosti te samostalno donosili odluke koje akcije poduzeti i kada. Najčešći aspekti pametne zgrade su komfor, sigurnost i energetska učinkovitost koji omogućavaju sigurnije i ugodnije stanovanje te smanjenje troškova. Za implementaciju ovakvih sustava koristi se agentno programiranje gdje su uređaji predstavljeni kao agenti te komunicirajući čine jedan višeagentni sustav koji je sposoban obavljati kompleksnije zadatke

Obzirom da je svakoj zgradi, bilo poslovnoj ili stambenoj, potrebna električna energija za napajanje različitih vrsta uređaja, u ovom radu fokus će biti na uštedi električne energije kroz prilagođavanje potrošnje. Osim što takva pametna zgrada pridonosi uštedi na strani korisnika, ona pomaže i u očuvanju okoliša. Većina elektrana u današnjem svijetu je još uvijek pogonjena fosilnim gorivima koja jako pridonose zagađenju. Elektrane, također, moraju zadovoljiti vršne vrijednosti, koje se događaj vrlo rijetko, kako ne bi došlo do ispada te u većini slučajeva nepotrebno proizvode električnu energiju te dodatno zagađuju okoliš. Implementacijom pametne zgrade koja će balansirati potražnjom električne energije, ili čak koristiti obnovljive izvore, moguće je smanjiti vršnu potražnju, a samim time i troškove održavanja elektrane, proizvodnje električne energije te na kraju i uštediti na računu za struju.

U prvom poglavlju ovog rada bit će razrađena teorija oko agenta. Bit će objašnjeno što je to agent i što ga čini inteligentnim. Zatim će biti objašnjeno u kakvim sve okolinama mogu funkcionirati agenti te koje su karakteristike pojedine okoline. Bit će obrađena i podjela agenata prema arhitekturi te će biti objašnjene moguće arhitekture agenata. Za kraj će biti objasnjeno na koje sve načine može biti implementirana funkcija agenata za odabir najbolje akcije.

Sljedeće poglavlje pokrit će teoriju o višeagentnim sustavima. Bit će objašnjeno kako se agenti povezuju u višeagentni sustav, kako komuniciraju te koje su karakteristike višeagentnog sustava. Zatim će biti napravljena usporedba višeagentnih sustava s ekspertnim sustavima i s objektima, to jest. objektno orijentiranim programiranjem.

Sljedeća dva poglavlja vezat će se uz dvije ključne stvari za postizanje električne učinkovitosti pametne zgrade. Prvo će biti obrađena i objašnjena pametna mreža te pametno

brojilo kao ključni dio pametne mreže s korisničke strane, a zatim će biti objašnjeno funkcioniranje DR programa. Bit će nabrojane i objašnjene vrste DR programa te koje su karakteristike i ciljevi takvog pristupa pri opskrbi električne energije.

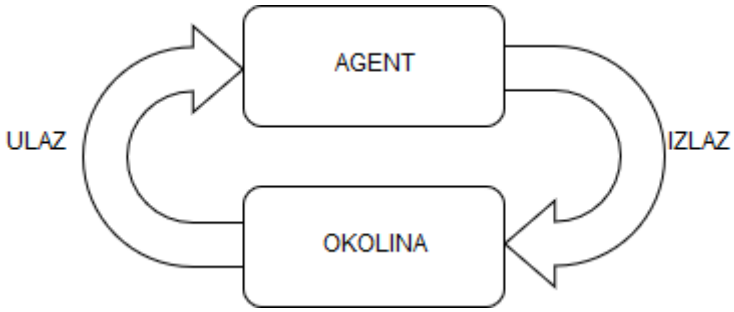
Zadnje poglavlje prikazat će implementaciju sustava pametne zgrade koja je dio DR programa. Prvo će biti objašnjen programski okvir JADE koji je korišten za implementaciju. Zatim će biti opisan sam sustav, njegova struktura i funkcioniranje. Nakon toga slijedi prikaz rada sustava te analiza rezultata i usporedba s običnom zgradom.

2. Inteligentni agent

Agent je fizički ili programski autonomni entitet, smješten u okolinu iz koje, putem senzora, prima određene informacije na temelju kojih donosi odluku i, putem aktuatora, poduzima akciju nad tom okolinom, kako bi postigao određeni cilj. Autonomnost znači da agent može samostalno djelovati, prilagođavati se i upravljati svojim unutarnjim stanjem kada se dogodi promjena u njegovoj okolini.

Kako bi neki agent mogli smatrati inteligentnim agentom, on mora imati fleksibilno ponašanje. Fleksibilnost agenta očituje se kroz [1]:

- **Reaktivnost** – Agent ima kontinuiranu interakciju s okolinom te pravovremeno reagira na promjene u njoj.
- **Proaktivnost** – Sposobnost agenta da preuzme inicijativu te da generira ciljeve i djeluje racionalno kako bi te ciljeve ostvario. Agent, dakle, nije pogonjen isključivo događajima u okolini već je pogonjen ciljevima.
- **Društvenost** – Sposobnost interakcije i suradnje s drugim agentima.



Slika 1. Agent i okolina (Wooldrige [1])

2.1. Okolina

Ranije je spomenuto da se agent nalazi u okolini koju promatra i iz koje prima podatke te na koju djeluje. Okolina ne mora nužno biti fizička, već može biti i programska.

Prema Russellu i Norvigu [1] okoline mogu biti :

1. **Potpuno primjetne ili djelomično primjetne** (engl. *fully observable or partially observable*) – Potpuno primjetna su ona u kojima agent ima pristup

kompletnom stanju okoline u svakom trenutku, u suprotnom je okolina djelomično primjetna.

2. **Pristupačne ili nepristupačne** (engl. *accessable or non-accessable*) – „Pristupačno okružje je okružje u kojem agent može dobiti potpune, točne i pravovremene informacije“ [2]. Pristupačnije okruženje znači lakšu implementaciju agenta.
3. **Determinističke ili stohastičke** (engl. *deterministic or non-determinitic*) – Odnosi se na predvidivost rezultata akcije. Kod determinističke okoline rezultat, tj. Sljedeće stanje okoline se može predvidjeti dok je to kod stohastičkih okolina nemoguće jer postoje i drugi faktori koji utječu na rezultat.
4. **Dinamičke ili statičke** (engl. *dynamic or static*) – Odnosi se na promjene u okolini koje nisu posljedica akcije agenta. Ukoliko se promjene mogu dogoditi samo kao posljedica akcije agenta, okolina je statička, u suprotnom je dinamička.
5. **Epizodične ili ne epizodične** (engl. *episodic or non-episodic*) – Epizodična okolina je ona okolina u kojoj performanse agenta ovise o određenom broju diskretnih epoha, a performanse i iskustva u sljedećoj epohi ne ovise o akcijama iz prethodnih epoha.
6. **Diskretne ili kontinuirane** (engl. discrete or continuous) – Diskretne su okoline u kojima postoji konačan broj akcija i podražaja (npr. šah) te bi se takvi okolinama moglo upravljati nekom vrstom tablicom znanja.
7. **Jednoagentne ili višeagentne** (engl. *single-agent or multi-agent*) – Ukoliko u okolini djeluje samo jedan agent, okolina je jednoagentna, ukoliko ih je više, okolina je višeagentna.
8. **Poznate ili nepoznate** (engl. *known or unknown*) – U poznatom okruženju agenti znaju koje akcije trebaju poduzeti dok u nepoznatom trebaju učiti o okolini kako bi mogli provesti akciju.

2.2. Arhitektura

Anthony, Chin, Gan, Alfred i Lukose [3] objašnjavaju arhitekturu agenta kao temelj agentovog mehanizma za rezoniranje (engl. *reasoning*) te kao svojevrsni nacrt (engl. *blueprint*) za izgradnju agenta kao što klasa u objektno orijentiranom programiranju služi za izgradnju objekta. Također navode kako je arhitektura mozak agenta koji određuje kako su znanja/informacije predstavljeni unutar agenta, ali i određuje akcije koje bi agent trebao poduzeti temeljem mehanizma za rezoniranje.

Anthony, Chin, Gan, Alfred i Lukose [3], u grubo, dijele arhitekturu u tri grupe, klasična, kognitivna i semantička. U klasičnu arhitekturu, smještaju agente bazirane na logici (engl. *logic-based agents*), reaktivne agente, BDI (engl. *Belief-Desire-Intention*) agente te slojevite (hibridne) agente (engl. *layered (hybrid) agents*).

Wooldrige [2], s druge strane, dijeli arhitekture na apstraktne i konkretne. Pod apstraktnu arhitekturu ubraja čisto reaktivne agente (engl. *purely reactive agents*), agente s percepcijom te agente sa stanjima, a pod konkretnu arhitekturu smješta agente bazirane na logici, reaktivne agente, BDI agente te slojevite agente.

2.2.1. Apstraktna arhitektura

Prema Wooldrigeu [2], apstraktni prikaz agenta može se lako formalizirati ako pretpostavimo da skup $\boldsymbol{S = \{s_1, s_2, \ldots\}}$ predstavlja stanja okružja u kojem se agent nalazi, a skup $\boldsymbol{A = \{a_1, a_2, \ldots\}}$ predstavlja akcije koje agent može izvesti. Agent tada možemo prikazati funkcijom

$$\text{akcija} = S^* \rightarrow A$$

Nedeterminističko ponašanje okoline može se prikazati funkcijom

$$\text{okolina} = S \times A \rightarrow \varphi(S)$$

Ta funkcija uzima trenutno stanje okoline $\boldsymbol{s \in S}$ i akciju $\boldsymbol{a \in A}$ koju je izveo agent te ih mapira u skup stanja okoline ***okolina(s, a)*** koja mogu proizaći kada se nad stanjem s izvrši akcija a.

Interakcija agenta i okoline može se prikazati kao povijest koja je slijed:

$$p{:}s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} s_3 \xrightarrow{a_3} \ldots \xrightarrow{a_{i-1}} s_i \xrightarrow{a_i} \ldots$$

gdje je s_0 početno stanje agenta, a_i i-ta akcija koju je agent izveo te s_i i-to stanje okoline. Ukoliko je agent predstavljen funkcijom $akcija = S^* \rightarrow A$, a okolina funkcijom $okolina = S \times A \rightarrow \varphi(S)$, tada će navedeni slijed p predstavljati moguću povijest agenta u okolini ako i samo ako vrijede sljedeća dva uvjeta:

$$\forall \in N,\ a_i = akcija((s_0, s_1, \ldots, s_i))$$

$$\forall \in N \text{ takav da } i > 0,\ s_i \in okolina(s_{i-1}, a_{i-1})$$

2.2.1.1. Čisto reaktivni agenti

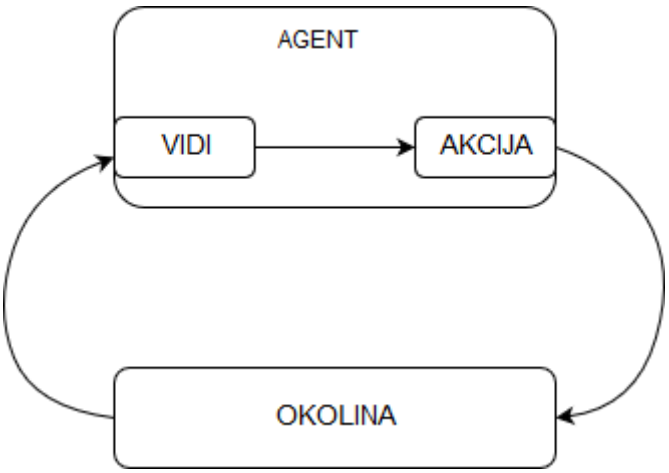
Čisto reaktivni agenti su agenti koji donose odluke isključivo bazirane na sadašnjem stanju bez uzimanja prošlosti u obzir. Takvi agenti mogu se prikazati funkcijom

$$akcija = S \rightarrow A$$

Termostat je jedan od primjera čisto reaktivnih agenata. Okolina termostata može biti u dva stanja, „Prehladno“ ili „Temperatura u redu“ pa funkcija akcije izgleda ovako:

$$akcija(s) = \begin{cases} \text{uključi grijanje, ako s = temperatura u redu} \\ \text{isključi grijanje, u suprotnom} \end{cases}$$

2.2.1.2. Percepcija



Slika 2. Percepcija i okolina (Wooldrige [2])

Iz Slike 2 vidljivo je da se agent sastoji od funkcije ***vidi*** pomoću koje agent promatra okolinu i funkcije ***akcija*** koja predstavlja donošenje odluke. Izlaz funkcije ***vidi*** je percepcija.

Ako je ***P*** ne prazan skup percepcija tada je funkcija ***vidi*** funkcija koja mapira stanje okoline na percepcije:

$$vidi = S \rightarrow P$$

Funkcija akcije tada mapira slijed percepcija na akcije

$$akcija = P^{*} \rightarrow A$$

2.2.1.3. Agenti sa stanjem

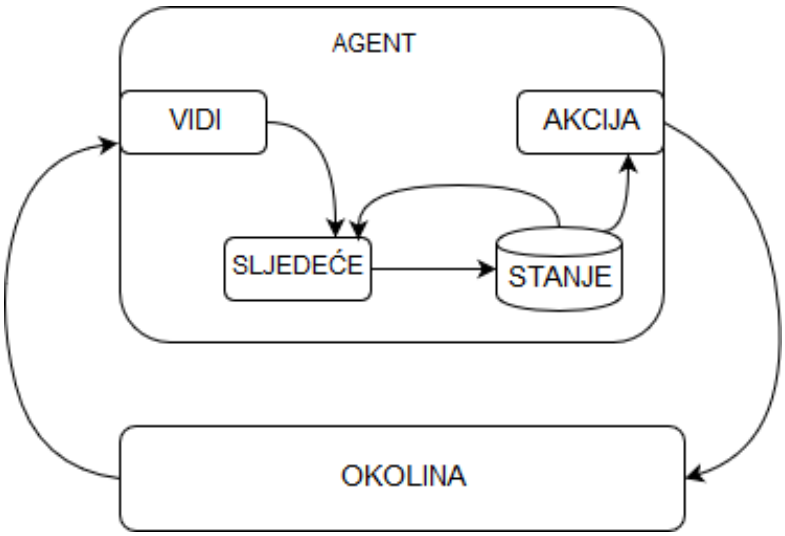
Agenti sa stanjem pamte informacije o stanju okoline te ih koristi kod donošenja odluke. Percepcijska funkcija vidi ostaje ista (vidi funkciju ***vidi*** u poglavlju 2.2.1.2).

Ako s ***I*** označimo skup unutarnjih stanja agenta tada funkcija akcija mapira interna stanja na akcije:

$$akcija = I \rightarrow A$$

Agenti sa stanjem imaju jednu dodatnu funkciju, funkciju ***sljedeće***, koja mapira unutarnje stanje i percepciju na unutarnje stanje:

$$sljedeće = I \times P \rightarrow I$$



Slika 3. Agent sa stanjima (Wooldrige [2])

2.2.2. Konkretna arhitektura

Kod apstraktnih arhitektura spomenuto je kako agent ima funkciju kojom nekako odlučuje koju akciju izvesti no nije definirano kako konkretno ta funkcija može biti implementirana. Isto tako spomenuti su agenti sa stanjima no nije navedeno kako ta stanja mogu izgledati i biti spremljena. Konkretizacija tih dviju stvari bit će objašnjena kroz konkretne arhitekture agenata pod koje spadaju agenti bazirani na logici, reaktivni agenti, BDI agenti i slojeviti agenti.

2.2.2.1. Agenti bazirani na logici

Ova arhitektura naziva se još i arhitektura bazirana na simbolici (engl. *symbol-based*) jer se okolina i ponašanje agenta prikazuju i modeliraju pomoću simboličkog prikaza kojim se upravlja pomoću sintakse. Simboličkim prikazom definira se pomoću logičkih formula dok se upravljanje pomoću sintakse realizira kao logička dedukcija ili dokazivanje teorema.

Neka je ***L*** skup rečenica klasične logike prvog reda kojima se opisuje okolina, a ***D*** baza podataka koja sadrži sve informacije o okolini. Bazu podataka ***D*** možemo prikazati kao ***D = p(L)*** gdje ***p(L)*** predstavlja skup svih mogućih okolina. Interno stanje agenta je tada element iz ***D***. Neka su Δ, Δ_1, … elementi skupa ***D***, neka je ρ skup dedukcijskih pravila koja predstavljaju proces odlučivanja u agentu i neka je ϕ formula koju treba dokazati tada će $\Delta \vdash_\rho \phi$ značiti da se formula ϕ može dokazati iz baze Δ koristeći samo dedukcijsko pravilo ρ. Moguća stanja okoline predstavljena su skupom $S = \{s_1, s_2, \ldots\}$, a akcije koje agent može odabrati skupom $A = \{a_1, a_2, \ldots\}$.

Percepcijska funkcija vidi ima formu ***vidi = S → P***, funkcija sljedeće ***sljedeće = D x P → D***, a funkcija akcija ***akcija = D → A***.

Funkcija akcija može se prikazati sljedećim pseudo kodom koji je u potpunosti preuzet od Wooldrigea [2]:

```
1. funkcija akcija(Δ : D) : A
2.begin
3.    for each a ∈ A do
4.          if Δ⊢ρDo(a) then
5.                return a
6.          end-if
7.    end-for
```

```
8.    for each a ϵ A do
9.          if Δ∀ρ⌐Do(a) then
10.               return a
11.         end-if
12.   end-for
13.   return null
14. end function akcija
```

Agent iterira kroz sve akcije i pokušava dokazati formulu ***Do*** iz baze, koja je proslijeđena kao ulazni parametar funkcije, koristeći dedukcijska pravila ***ρ*** (linije 3-7). Ukoliko uspije, izvodi akciju, a ukoliko ne uspije ide na sljedeću petlju (linije 8-12). U toj petlji agent pokušava pronaći akciju koja nije eksplicitno zabranjena, odnosno akciju ***a ∈A*** takvu da ***⌐Do(a)*** ne može biti derivirano iz baze koristeći dedukcijska pravila ***ρ***. Ukoliko ne postoji ni takva akcija, vraća se specijalna akcija null.

Iako je ova arhitektura prilično jednostavna i elegantna, postoje mnogi problemi vezani uz nju. Teško je prevesti informacije okoline u simboličku informaciju, pogotovo ako se radi o kompleksnoj okolini. Također, jako je teško, a ponekad i nemoguće navesti sva pravila za sve situacije koje će agent doživjeti u okolini. U ovoj arhitekturi proces donošenja odluke bazira se na kalkulativnoj racionalnosti (engl. *calcualtive rationality*), tj. na pretpostavci da se okolina neće značajno promijeniti tijekom vremena, što naravno nije realno. Stoga je upitno kako će agent reagirati u dinamičnoj ili vremenski ograničenoj okolini.

2.2.2.2. Reaktivni agenti

Reaktivni agenti reagiraju na promjene u okolini bez da o njima rezoniraju. Najpoznatija reaktivna arhitektura je Brooksova arhitektura supsumpcije (engl. *subsumption architecture*)

Dvije su važne karakteristike arhitekture supsumpcije. Arhitektura je implementirana kao konačni automat (engl. *finite state machine*) Prva je da se donošenje odluke agenta realizira kroz skup ponašanja koja izvršavaju zadatke. Svako ponašanje može se shvatiti kao zasebna funkcija akcija koja mapira percepcijski ulaz direktno na akciju, bez simboličke reprezentacije i bez simboličkog rezoniranja ***situacija → akcija***.

Druga karakteristika je da se više ponašanja može okinuti u isto vrijeme. Kako bi se odlučilo koja od tih akcija će se izvršiti, Brooks uvodi hijerarhiju supsumpcije tako da se ponašanja slažu u slojeve. Viši slojevi predstavljaju više općenita, apstraktna ponašanja te

imaju manji prioritet. Što je ponašanje niže u sloju, ima veći prioritet. Niži slojevi tako sprječavaju ili blokiraju (engl. *to inhibit*) izvršavanje ponašanja viših slojeva.

Za pojašnjenje ove arhitekture, Wooldrige [2] formira jednostavan model. U tom modelu funkcija vidi ostaje nepromijenjena dok se funkcija akcija realizira kao skup ponašanja zajedno sa relacijom sprječavanja (engl. *inhibition relation*) između tih ponašanja. Ponašanje se opisuje kao par ***(c, a)*** gdje je $\mathbf{c \subseteq P}$ skup svih percepcija koji se još naziva i uvjet, a $\boldsymbol{a \in A}$ je akcija. Ponašanje ***(c, a)*** će se okinuti kada je okolina u stanju $\boldsymbol{s \in S}$ ako i samo ako je $\boldsymbol{vidi(s) \in c}$. Skup svih takvih ponašanja je $\boldsymbol{Beh = \{(c, a) \mid c \subseteq P \text{ i } a \subseteq A\}}$. Ako je skup pravila ponašanja agenta ***R*** tada je ***Beh*** binarna relacija sprječavanja $\boldsymbol{\prec \subseteq R \times R}$. Ako su $\boldsymbol{b_1, b_2 \in \prec}$ tada $\boldsymbol{b_1 \prec b_2}$ znači da $\boldsymbol{b_1}$ sprječava $\boldsymbol{b_2}$, tj. da se $\boldsymbol{b_1}$ nalazi niže u hijerarhiji. Akcijska funkcija prema Wooldigeu [2] izgleda ovako:

```
1. funkcija akcija(p : P) : A
2. var fired : ℘(R)
3. var selected : A
4. begin
5.    fired := {(c,a) | (c,a) ∈ R and p ∈ c}
6.    for each (c,a) ∈ fired
7.          if ⌐(∃(c′,a′) ∈ fired such that (c′,a′) ≺ (c,a)) then
8.                return a
9.          end-if
10.   end-for
11.   return null
12. end function akcija
```

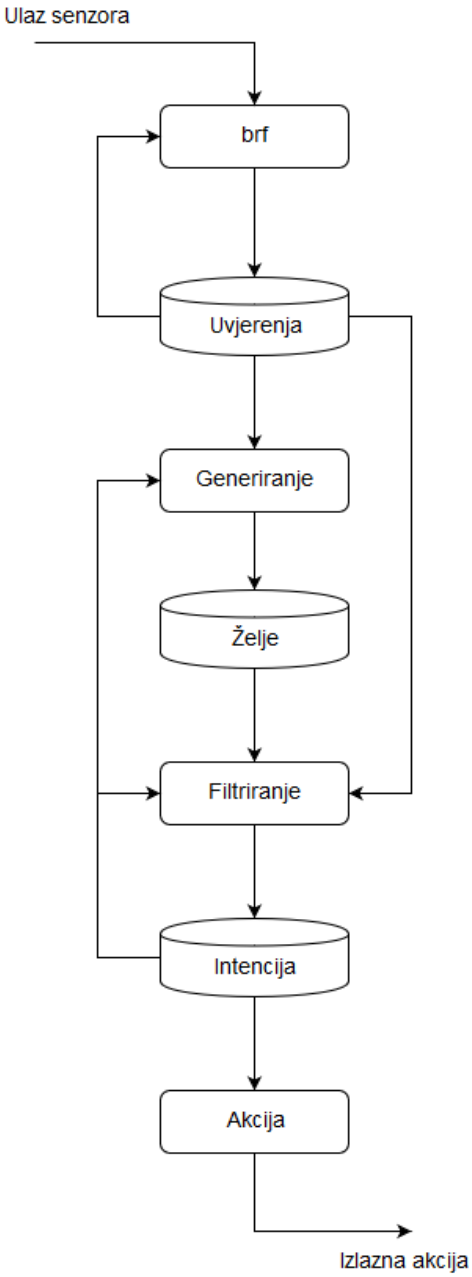
Prvo se računa set svih ponašanja koja su se okinula. Ako se nije okinulo ni jedno ponašanje vraća se `null`, a u suprotnom se provjerava svako ponašanje kako bi se utvrdilo postoji li ponašanje višeg prioriteta. Ukoliko nema ponašanja višeg prioriteta vraća se akcija a trenutnog ponašanja.

Prednosti ove arhitekture su jednostavnost, ekonomičnost, robusnost protiv neuspjeha i elegantnost, no postoje i neki temeljni problemi koji nisu riješeni. Jedan od problema je što čisto reaktivni agenti odlučuju na temelju lokalnih informacija, tj. unutarnjih stanja pa je teško zaključiti kako takav proces može uzeti u obzir druge informacije koje nisu lokalne. Teško je postići efekt učenja i poboljšanja performansi tijekom vremena. Također je teško i izgraditi takve agente zbog složene interakcije između slojeva, ali i zato što još uvijek nisu potpuno jasne veze između pojedinih ponašanja, okoline i ukupnog ponašanja.

2.2.2.3. BDI agenti

BDI agenti temelje se na praktičnom rezoniranju, procesu donošenja odluka usmjerenog prema akcijama. Praktično rezoniranje se sastoji od dva procesa:

- **Izbor ciljeva** – odlučivanje o tome što se želi postići
- **Izbor akcija** – odlučivanje kako se želi postići odabrane ciljeve



Slika 4. Shematski dijagram generičke BDI arhitekture (Wooldrige[2])

Slika 4. prikazuje generičku arhitekturu BDI agenta iz koje je vidljivo da se BDI agent sastoji od sljedećih sedam komponenti:

- **Skup trenutnih uvjerenja** (engl. *belief*) – informacije koje agent posjeduje o okolini
- **Funkcija revizije uvjerenja** (engl. *belief revision function* - brf) – prima percepcijski ulaz i trenutna uvjerenja i temeljem tih informacija određuje novi set uvjerenja
- **Funkcija generiranja opcija** – funkcija koja temeljem nakana (engl. *intention*) i uvjerenja generira opcije (želje, engl. *desire*)
- **Skup trenutnih opcija** – predstavlja akcije dostupne agentu
- **Funkcija filtriranja** – određuje nakane agenta temeljem njegovih trenutnih uvjerenja, želja i nakana
- **Skup trenutnih nakana** – predstavljaju nakane koje se agent obvezao postići

Neka je ***Bel*** skup svih uvjerenja, ***Des*** skup svih želja, a ***Int*** skup svih nakana. BDI agent može se prikazati kao uređena trojka ***(B, D, I)*** gdje je $B \subseteq Bel$, $D \subseteq Des$, a $I \subseteq Int$.

Funkcija revizije uvjerenja (***brf***) je mapiranje – $brf: \wp(Bel) \times P \rightarrow \wp(Bel)$

Funkcija za generiranje opcija je mapiranje – $opcija: \wp(Bel) \times \wp(Int) \rightarrow \wp(Des)$

Funkcija za filtriranje je mapiranje – $filtriraj: \wp(Bel) \times \wp(Int) \times \wp(Des) \rightarrow \wp(Des)$

Funkcija za filtriranje mora zadovoljiti sljedeću formulu:

$\forall B \in \wp(Bel), \forall D \in \wp(Des), \forall I \in \wp(Int), filter(B,D,I) \subseteq I \cup D$

Funkcija izvrši (engl. *execute*) vraća nakanu za izvršavanje akcije:

$execute: \wp(Int) \rightarrow A$

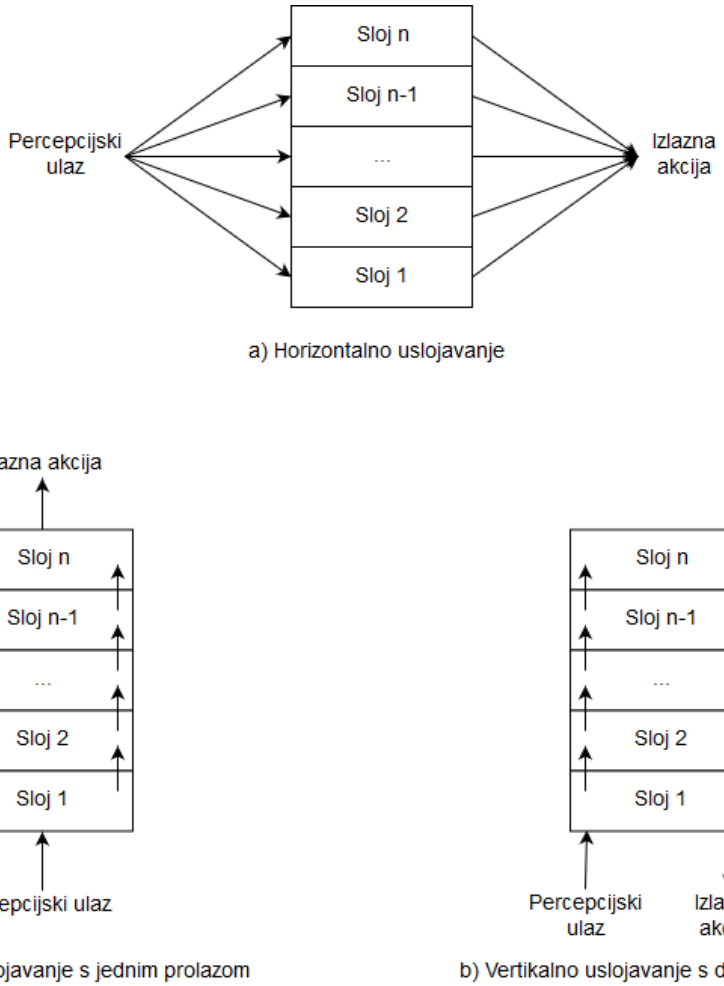
Funkcija akcija, $akcija = P \rightarrow A$, može se prikazati sljedećim pseudo kodom, preuzetim od Wooldrigea [2]:

```
1. funkcija akcija(p : P) : A
2. begin
3.     B := brf(B,p)
4.     D := opcija(D,I)
5.     I := filter(B,D,I)
6.     return izvrši(I)
7. end function akcija
```

Čisti i intuitivni dizajn te jasna funkcionalna dekompozicija prednosti su BDI arhitekture te omogućuju lakši razvoj agenta. Nedostatak je što nije jasno kako učinkovito implementirati funkcionalnost podsustava pa agent mora pronaći balans između obvezivanja (engl. *commitement*) i revizije. Kako navode Anthony, Chin, Gan, Alfred i Lukose [3], ako agent ne prestane s revizijom, može se dogoditi da pokušava postići nakane koje više nisu moguće ili valjane. Ako pak prečesto revidira može doći do rizika neizvršavanja nakana zbog nedovoljno vremena provedenog radeći na zadatku.

2.2.2.4. Slojeviti (hibridni) agenti

Kako agenti moraju biti sposobni za reaktivno i pro aktivno ponašanje, razvila se ideja da se ta dva ponašanja podijele u različite podsustave, a samim time i da se ti podsustavi organiziraju kao hijerarhija s interaktivnim slojevima (uslojavanje). Slojevita arhitektura kombinira prednosti agenata baziranih na logici i reaktivnih agenata, a istovremeno ublažuje probleme koji postoje u tim arhitekturama. Dvije su vrste slojevite arhitekture: horizontalno uslojavanje i vertikalno uslojavanje.



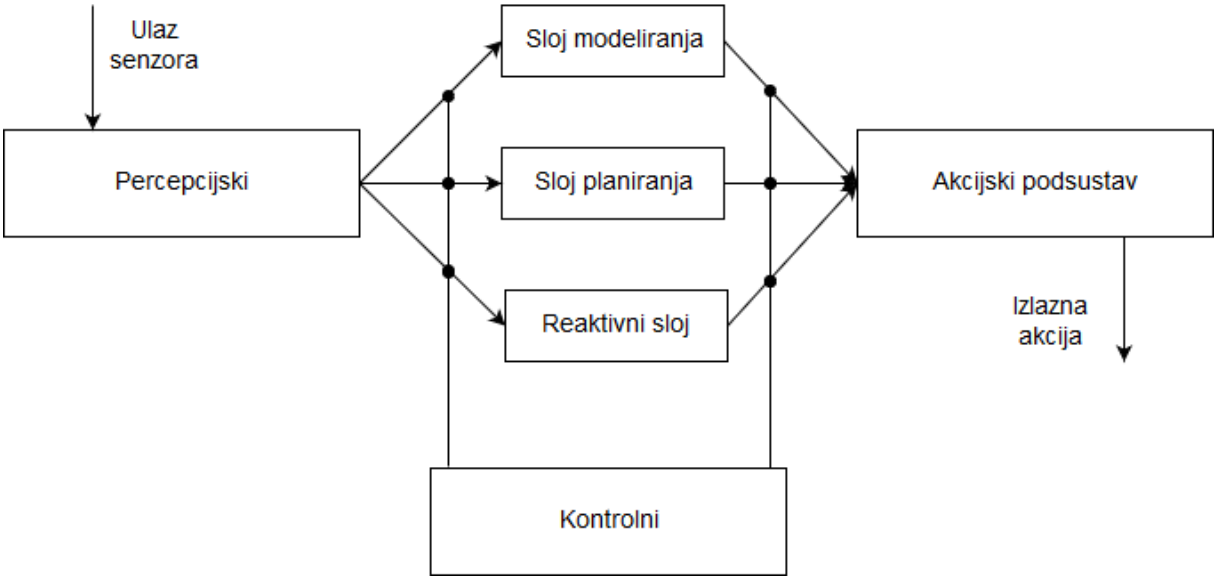
Slika 5. Vertikalna i horizontalna arhitektura (Wooldrige [2])

Kod horizontalnog uslojavanja svaki sloj prima ulazne informacije od senzora i predlaže izlazne akcije, tj. svaki sloj se ponaša kao zasebni agent. Prednost arhitekture je što je potrebno samo n slojeva da bi se postiglo n tipova ponašanja no problem je što treba uzeti u obzir m^n interakcija, pri čemu m predstavlja broj prijedloga mogućih akcija.

Primjer horizontalne slojevite arhitekture je *TouringMachines* koji se sastoji od tri sloja koji istodobno i nezavisno mapiraju percepcije u akciju:

1. **Reaktivni sloj** (engl. *reactive layer*) – implementiran kao skup pravila za određene situacije s ciljem da pruži neposredan odgovor u slučaju promjene okoline
2. **Sloj planiranja** (engl. *planning layer*) – donosi odluku što će agent raditi kako bi postigao ciljeve ili nakane ili izvršio zadatke

3. **Sloj modeliranja** (engl. *Modeling layer*) – predstavlja različite entitete u svijetu, uključujući i samog agenta i druge agente. Predviđa konflikte među agentima i generira ciljeve kako bi se ti konflikti razriješili



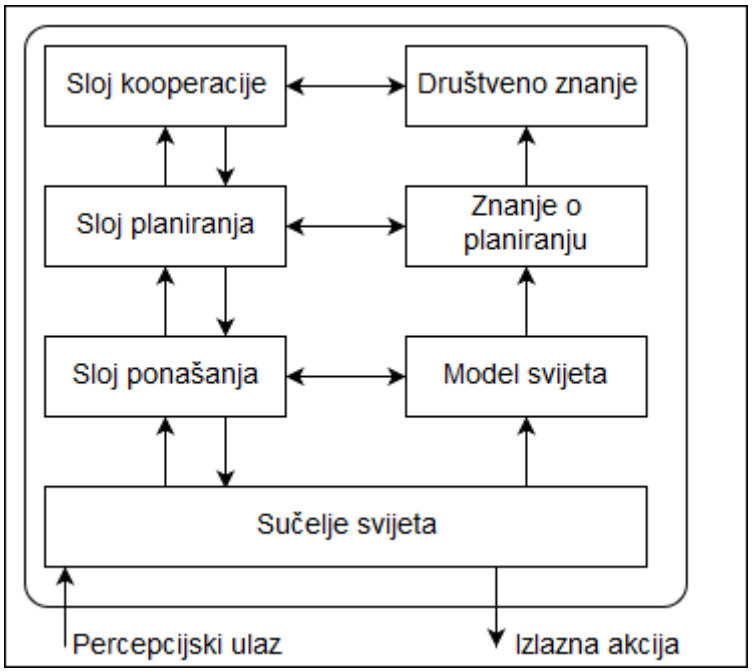
Slika 6. TouringMachines: Arhitektura s horizontalnim uslojavanjem (Wooldrige [2])

Kod vertikalne arhitekture ulaznu informaciju dobivenu iz senzora obrađuje maksimalno jedan sloj, a isto tako i izlaznu akciju odrađuje maksimalno jedan sloj što rješava ranije spomenutu problem velikog broja interakcija kod horizontalnog uslojavanja. Broj interakcija kod vertikalnog uslojavanja je $\boldsymbol{m^2 (n - 1)}$. Najveći nedostatak je to što ispad jednog sloja uzrokuje pad cijelog sustava.

Postoje dva tipa vertikalnog uslojavanja:

1. **Arhitektura jednog prolaza** (engl. *one-pass architecture*) – prvi sloj dobiva ulazne informacije od senzora te ih prosljeđuje do zadnjeg sloja koji generira akciju
2. **Arhitektura dva prolaza** (engl. *two-pass architecture*) - prvi sloj dobiva ulazne informacije od senzora te ih prosljeđuje do zadnjeg sloja, a nakon toga se informacije vraćaju od zadnjeg sloja do prvog koji generira akciju.

Primjer vertikalnog uslojavanja s dva prolaza je InteRRap čija arhitektura je prikazana na Slici 7.



Slika 7. InteRRap: Vertikalna arhitektura s dva prolaza (Wooldrige [2])

2.3. Program agenta

Program agenta je implementacija akcijske funkcije. Iako je funkcija akcija definirana kao mapiranje slijeda percepcija u akcije, program agenta prima samo jednu, trenutnu, percepciju te sam izgrađuje slijed, ukoliko je to potrebno.

Ovisno o načinu kako je funkcija akcija implementirana te mogućnostima i razini inteligencija, programe agenata moguće je grupirati u pet klasa:

- Jednostavni refleksivni agenti (engl. *Simple reflex agenst*)
- Refleksivni agenti temeljeni na modelu (engl. *Model-based reflex agents*)
- Agenti temeljeni na ciljevima (engl. *Goal-based agents*)
- Agenti temeljeni na korisnosti (engl. *Utility-based agents*)
- Agenti sa sposobnošću učenja (engl. *Learning agents*)

2.3.1. Jednostavni refleksivni agenti

Jednostavni refleksivni agenti su najjednostavniji agenti. Oni odabiru akciju samo na temelju trenutne percepcije ne obazirući se na prošlost. Russell i Norvig [1] kao primjer navode automatizirani automobil koji reagira na kočenje automobila ispred. Ukoliko automobil ispred zakoči, njegova svijetla za kočenje se upale, automatizirani automobil to primijeti te također započinje kočenjem.

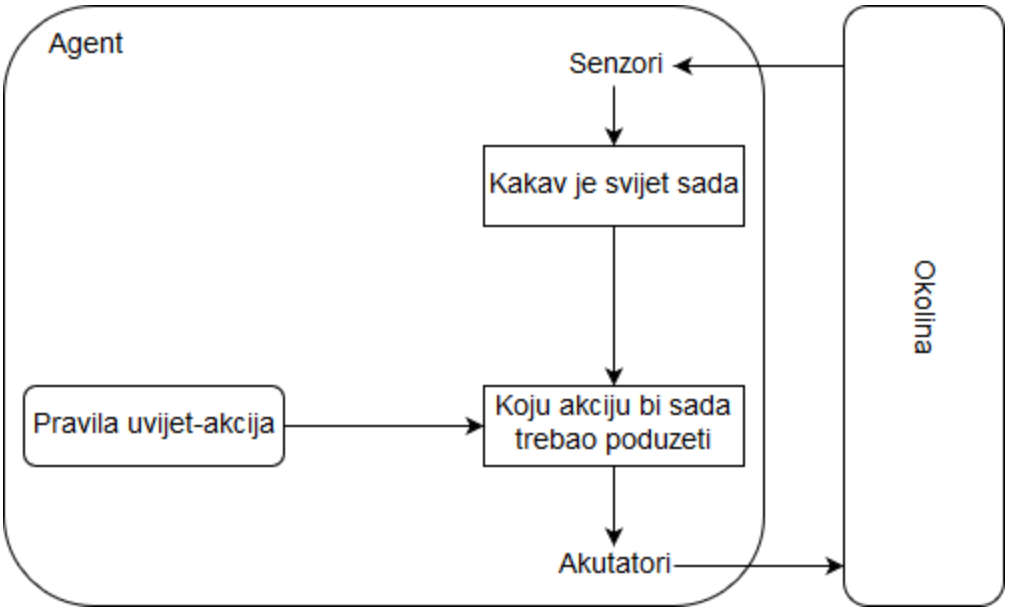
Prednost ovih agenata je njihova jednostavnost, no mana je to što imaju limitiranu inteligenciju. Ukoliko u gornjem primjeru automobil ispred nema svjetla za kočenje automatizirani automobil neće reagirati.

Program jednostavnog refleksivnog agenta, prema Russellu i Norvigu [1], može se prikazati sljedećim pseudo kodom:

```
function SIMPLE-REFLEX-AGENT(percept) returns an action
      persistent: rules, a set of condition-action rules

      state ← INTERPRET-INPUT(percept)
      rule ← RULE-MATCH(state, rules)
      action ← rule.ACTION
      return action
```

Funkcija SIMPLE-REFLEX-AGENT prima percepciju, tj. trenutno stanje okoline te vraća akciju temeljenu na pravilima uvjet-akcija. INTERPRET-INPUT prima trenutno stanje te kreira apstraktni opis trenutnog stanja koji se zajedno sa skupom pravila predaje metodi RULE_MATCH. RULE_MATCH tada vraća prvo pravilo koje odgovara opisu trenutnog stanja te se iz tog pravila uzima akcija koju agent izvršava.



Slika 8. Jednostavni refleksivni agent (Russell i Norvig [1])

2.3.2. Refleksivni agenti temeljeni na modelu

Refleksivni agenti temeljeni na modelu održavaju interno stanje koje ovisi o prošlim percepcijama. Temeljem tih stanja može pretpostaviti određene aspekte trenutnog stanja okoline. Kako bi agent ažurirao unutarnje stanje potrebne su mu dvije vrste znanja, znanje o tome kako se svijet mijenja neovisno o agentu te znanje o tome kako agent utječe na promjenu svijeta. Navedena znanja koja predstavljaju kako svijet funkcionira nazivaju se još i model svijeta.

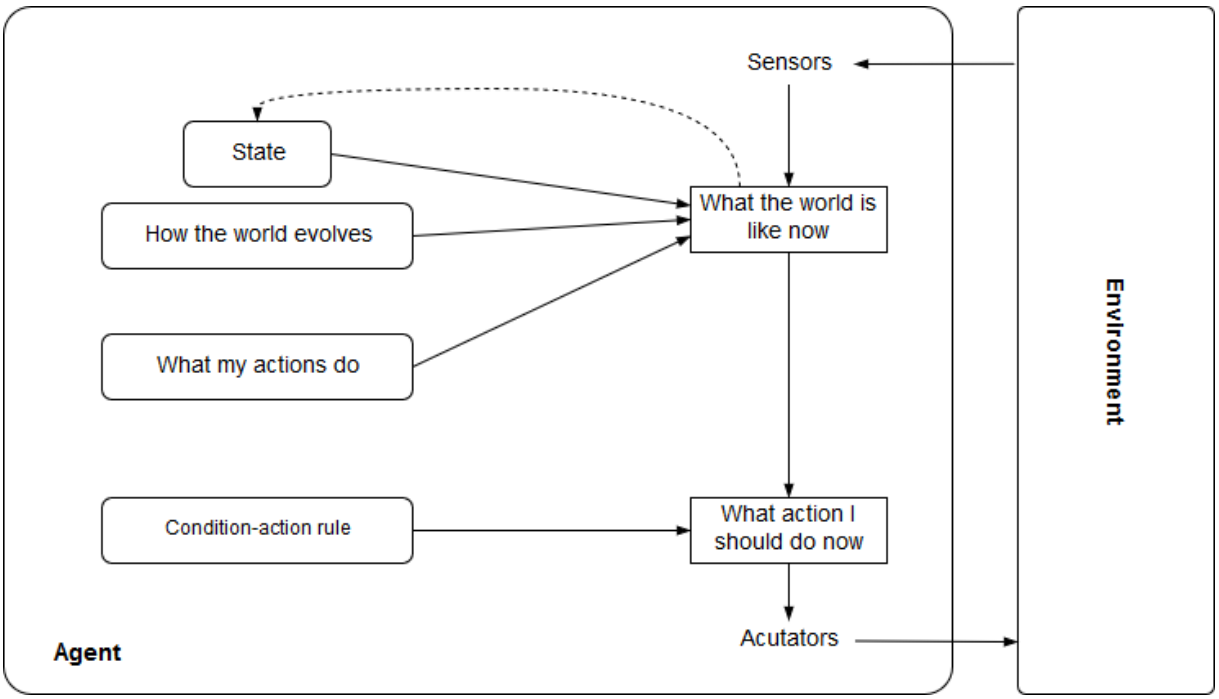
Ovi agenti su kompliciranu od jednostavnih refleksivnih agenata no zbog znanja koja imaju o tome kako svijet funkcionira, mogu donijeti zaključke te odrediti neke aspekte svijeta koje ne mogu dobiti samo pomoću promatranja svijeta.

Program refleksivnog agenta temeljenog na modelu, prema Russellu i Norvigu [1], može se prikazati sljedećim pseudo kodom:

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
      persistent: rules, a set of condition-action rules
model, a description how the next state depends on current state and action

      state ← INTERPRET-INPUT(percept)
      rule ← RULE-MATCH(state, rules)
      action ← rule.ACTION
      return action
```

Za razliku od Jednostavnih refleksivnih agenata, umjesto INTERPRET-INPUT metode postavljena je UPDATE-STATE metoda koja prima trenutni koncept stanja koje agent ima o svijetu, zadnju akciju koju je izveo, trenutnu percepciju i model. Temeljem tih ulaza metoda generira novi opis unutarnjeg stanja agenta. Ostale su funkcije iste kao i kod jednostavnog refleksivnog agenta te rade na isti način.



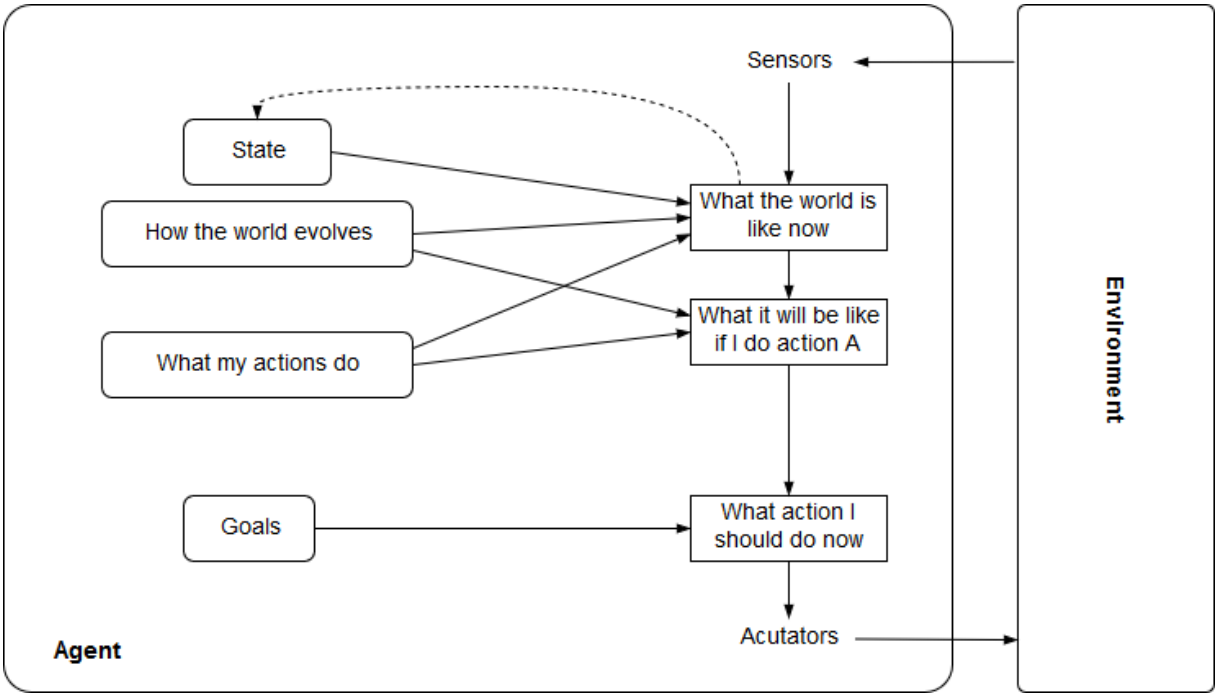
Slika 9. Refleksivni agent temeljen na modelu (Russell i Norvig [1])

2.3.3. Agenti temeljni na ciljevima

Unutarnja stanja ponekad nisu dovoljna da bi agent odlučio što učiniti, stoga mu je potrebno zadati cilj kako bi agent znao što se želi postići. Russell i Norvig [1] ponovno upotrebljavaju taksi kao primjer. Taksi mora odlučiti kuda će otići na raskrižju, a u tome mu pomaže postavljeni cilj, tj. odredište na koje putnik želi doći.

Donošenje odluka ovdje se jako razlikuje od pravila uvjet-akcija jer agent razmatra što će se dogoditi u budućnosti ako poduzme neku akciju i je li taj rezultat zadovoljavajuć, tj. postiže li zadani cilj.

Iako su ovi agenti dosta kompleksniji, njihova prednost je fleksibilnost. Za razliku od jednostavnih refleksivnih agenata gdje je potrebno raspisati sve uvjete i vezati ih uz akcije, ovi agenti sami ažuriraju znanja i kreiraju ponašanja kako bi postigli zadani cilj.



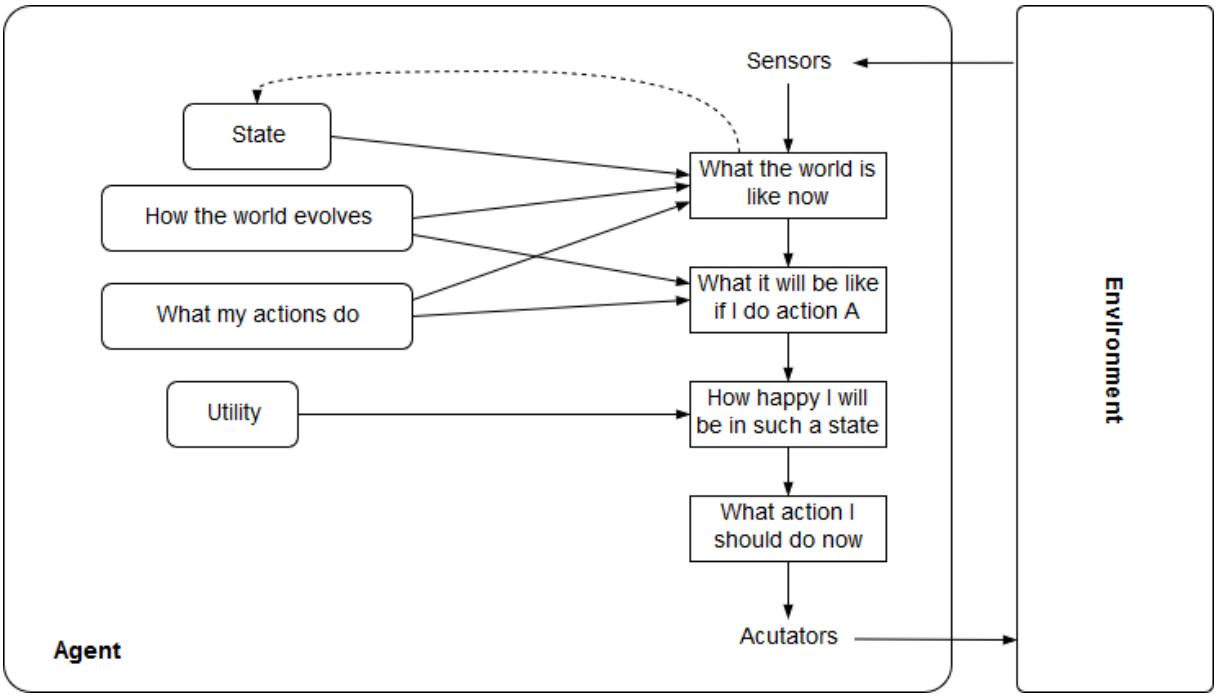
Slika 10. Agent na temelju modela, na temelju ciljeva (Russell i Norvig [1])

2.3.4. Agenti temeljeni na korisnosti

Ciljevi pružaju mogućnost agentu da odredi je li „sretan“ ili „nesretan“ stanjem koje će određena akcija prouzročiti. Obično se isti cilj može postići na više načina stoga je potrebno nekako definirati koji slijed akcija će učiniti agent „više sretnim“. Ako se neko stanje preferira više od drugog stanja, tada kažemo da ono ima veću korisnost za agenta.

Prema Russellu i Norvigu [1], korisnost je funkcija koja mapira stanje u broj, koji definira razinu sreće. Ukoliko postoje konfliktni ciljevi, od kojih samo neki mogu biti postignuti, funkcija korisnosti specificira određeni kompromis. Ukoliko postoji više ciljeva, od kojih ni jedan nije moguće postići sa sigurnošću, funkcija korisnost pruža način na koji se vjerojatnost uspjeha može mjeriti s važnošću ciljeva.

Prednosti ovih agenata su fleksibilnost i mogućnost učenja, no nedostatak je što može biti jako kompliciranо implementirati funkciju korisnosti.



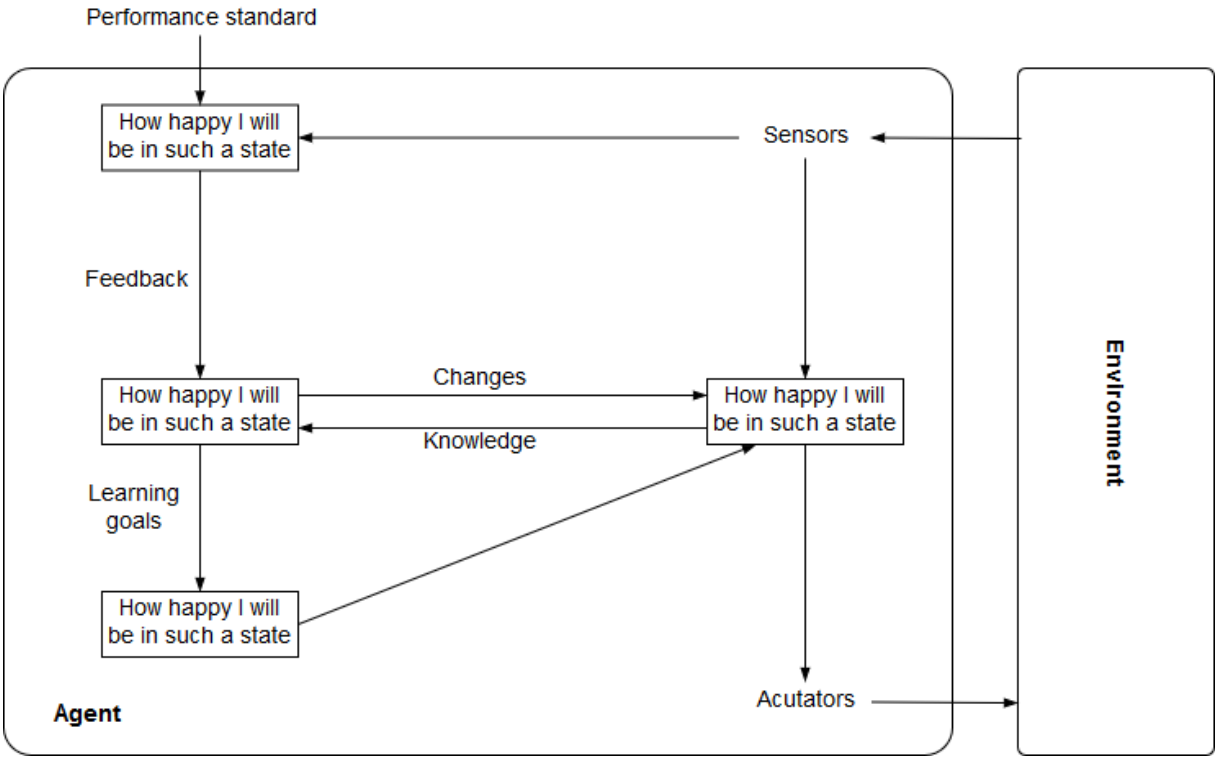
Slika 11. Agent temeljen na modelu, temeljen na korisnosti (Russell i Norvig [1])

2.3.5. Agenti sa sposobnošću učenja

Agenti sa sposobnošću učenja sastoje se od četiri komponente: element učenja (engl. *learning element*), element performanse (engl. *performance element*), kritika (engl. *critic*) i generator problema (engl. *problem generator*). Element performanse uzima percepcije te odlučuje o akciji koja će se izvesti, a element učenja je zadužen za poboljšanja na temelju povratne informacije dobivene od strane kritičkog elementa. Kritika, dakle, daje informaciju agentu o tome koliko je uspješan, tj. koliko je zadovoljavajuće to što je napravio, te na temelju nje, element učenja modificira element performanse kako bi u budućnosti imao bolje rezultate. Generator problema predlaže (istraživačke) akcije kako bi se došlo do novih saznanja i kako bi se pronašle bolje akcije koje element performanse može poduzeti.

Russell i Norvig [1] ponovno koriste primjer automatiziranog taksija kako bi objasnili elemente ovog agenta. Automatizirani taksi izlazi na cestu i vozi pomoću elementa performanse, dok kritika promatra svijet i prosljeđuje informacije elementu učenja. Ukoliko se taksi u jednom trenutku prestroji preko tri trake, kritika će primijetiti reakcije drugih vozača, poput trubljenja, upotrebe neprimjerenog rječnika, naglog kočenja i slično, te će proslijediti informaciju elementu učenja. Element učenja će, iz iskustva, kreirati novo pravilo koje kaže kako ta akcija nije bila dobra te će to pravilo umetnuti u element performanse i time ga poboljšati. Generator problema može identificirati različita područja koja je potrebno

poboljšati, kao na primjer kočenje, te može predložiti eksperiment kočenja na različitim podlogama ili u različitim uvjetima.



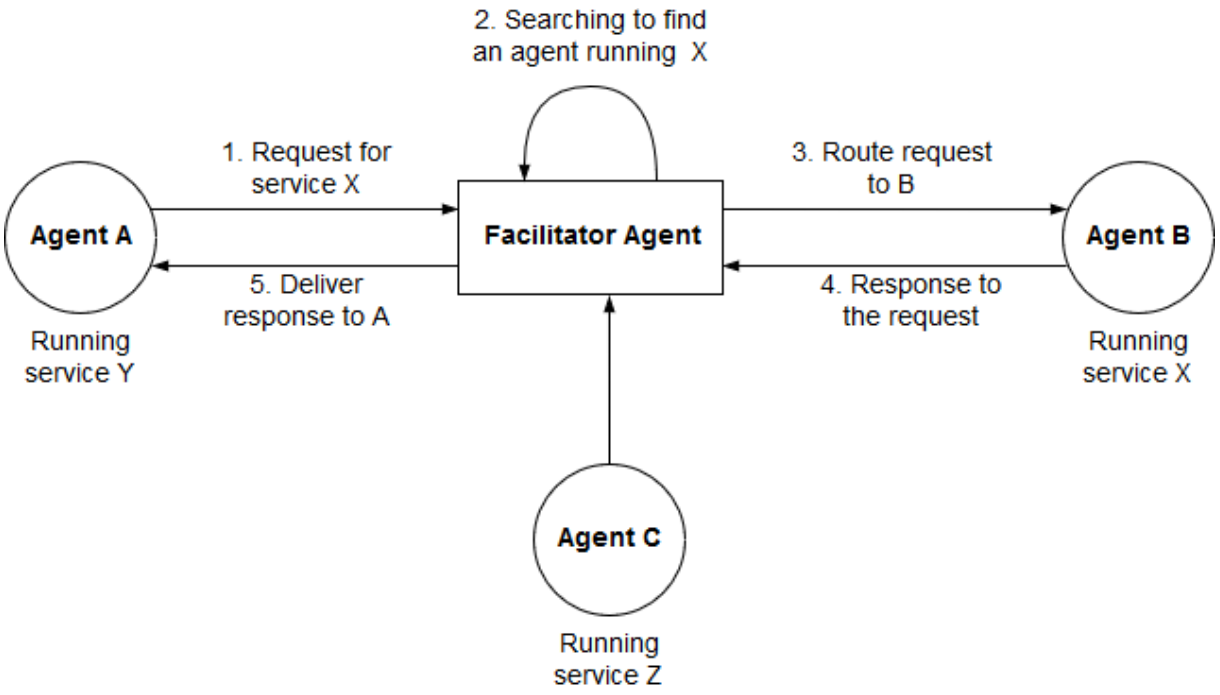
Slika 12. Općeniti agent sa sposobnošću učenja (Russell i Norvig [1])

3. Višeagentni sustavi

Kao što sami naziv kaže, višeagentni sustavi su sustavi koji se sastoje od više interaktivnih inteligentnih agenata. U današnje vrijeme, kada su računala svuda oko nas, sve više i više stvari se pokušava informatizirati i automatizirati, a problemi koje se pokušava riješiti postaju toliko kompleksni da ih monolitni sustavi ili samostalni agenti ne mogu sami riješiti. Stoga se kompleksni problemi razdvajaju na manje zadatke koji se dodjeljuju agentima te svaki agent temeljem određenih ulaznih informacija odlučuje kako riješiti pridijeljeni zadatak. Kako navode Dorri, Jurdak i Kanhere [4], glavne odlike višeagentnog sustava su učinkovitost, niski troškovi, fleksibilnost i pouzdanost. Efikasnost leži u već ranije spomenutom razdvajanju zadatka na više manjih zadataka koji se dodjeljuju agentima. Raspodjela također donosi i smanjenju troškova jer se potrošnja energije, procesiranje i slično također dijeli na agente što ispada jeftinije nego kada bi cijeli problem trebala riješiti jedna moćna jedinica. Pouzdanost je također vezana uz ovakav pristup distributivnog rješavanja problema jer takav pristup omogućuje prebacivanje zadatka na druge agente u slučaju da jedan agent zakaže. Fleksibilnost predstavlja mogućnost agenta da riješi dodijeljeni zadatak bez obzira na razinu inicijalnih znanja.

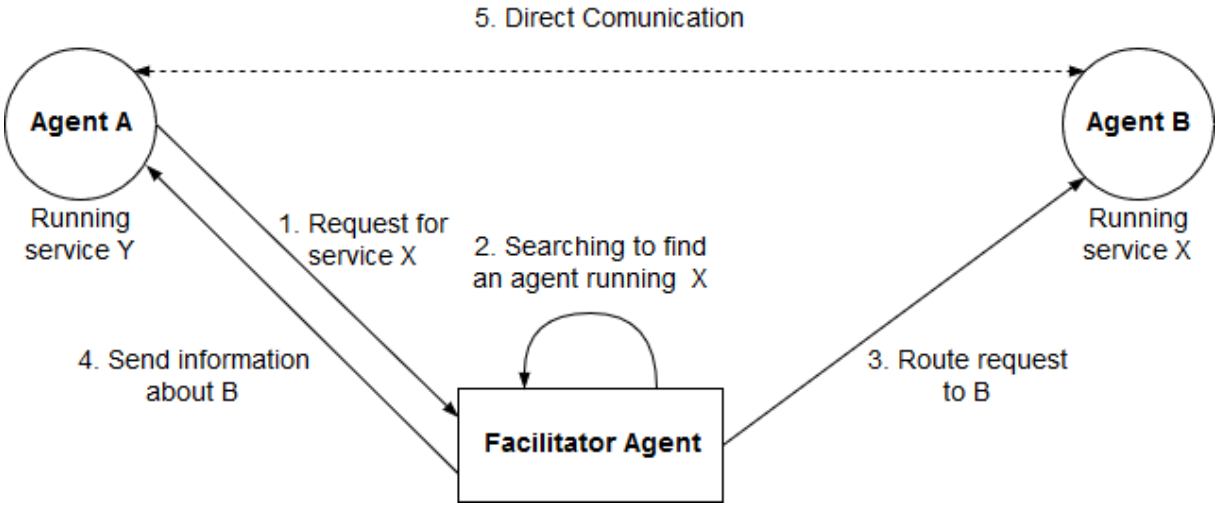
Kako bi se smanjila komunikacija i spriječilo preopterećenje svaki agent najčešće komunicira samo s izravnim susjedima. S druge strane, takav pristup uzrokuje povećanje komunikacije, ali i utrošenog vremena kada agent pokušava pronaći agenta koji mu može pružiti određeni servis ili funkciju. Kako bi se to riješilo, uvodi se jedan središnji agent (engl. *middle agent*) koji sadrži listu servisa koje pružaju agenti i kojeg agent kontaktira kada želi pronaći agenta za koji mu može pružiti željeni servis. Postoje dva načina implementacije središnjeg agenta:

1. **Voditelj** (engl. *facilitator*) – prima zahtjev od agenta koji traži servis i prosljeđuje ga agentu koji pruža servis te prima odgovor od agenta koji pruža servis i prosljeđuje ga agentu koji je tražio servis. Problem je što na ovaj način središnji agent predstavlja usko grlo što se može riješiti uvođenjem više voditeljskih agenata no oni onda moraju raditi sinkronizirano



Slika 13. Središnji agent, voditelj (engl. *Facilitator*; Dorri, Jurdak i Kanhere [4])

2. **Medijator** (engl. *mediator*) – za razliku od voditeljskog agenta, ovdje agent koji šalje zahtjev i agent koji pruža servis mogu komunicirati direktno što se može vidjeti na slici



Slika 14. Središnji agent, medijator (Dorri, Jurdak i Kanhere [4])

3.1. Karakteristike višeagentnog sustava

Dorri, Jurdak i Kanhere [4] navode osam važnih karakteristika višeagentnog sustava:

1. **Rukovođenje** (engl. leadership) – Višeagentni sustav može biti bez rukovoditelja i s rukovoditeljem. Ukoliko nema rukovoditelja svaki agent odlučuje samostalno koje će akcije poduzeti vodeći se ciljevima. Kod sustava s rukovoditeljem, rukovodeći agent odlučuje o akcijama za druge agente, a ti agenti razmjenjuju informacije kako bi saznali poziciju rukovoditelja. Rukovoditelj može biti unaprijed definiran ili odabran zajednički od strane agenata. Višeagentni sustav može imati mobilnog rukovoditelja ili grupu agenata koji se ponašaju kao rukovoditelji.
2. **Funkcija odlučivanja** (engl. decision function) – Ovisno o tome kako se mijenja izlaz iz funkcije odlučivanja kada se promijeni ulaz, višeagentne sustave dijelimo na linearne i ne linearne. Kod linearnih, odluka agenta je proporcionalna parametrima koje prima iz okoline. Kod ne linearnih sustava, odluka agenta nije proporcionalna parametrima iz okoline jer je i sam ulaz nelinearan.
3. **Heterogenost** – Višeagentni sustavi mogu biti homogeni i heterogeni. Homogeni se sastoje od agenata koji imaju iste karakteristike i funkcionalnosti dok kod heterogenih sustava agenti imaju različite karakteristike i funkcionalnosti.
4. **Parametri dogovaranja** (engl. *agreement parameters*) – U nekim višeagentnim sustavi agenti se moraju dogovoriti oko metrika. Obzirom na broj metrika, višeagentne sustave dijelimo na sustave prvog reda, drugog reda i višeg reda. Sustavi prvog reda moraju se dogovoriti oko jedne metrike, a kod drugog reda oko dvije metrike. Kod sustava višeg reda dogovor je postignut kada se metrike agenata i njihovi derivati višeg reda približe zajedničkoj vrijednosti.
5. **Uzimanje kašnjenja u obzir** (engl. *delay consideration*) – Višeagentne sustave možemo podijeliti ovisno o tome uzimaju li u obzir izvore kašnjenja ili ne. Stoga se dijele na sustave s kašnjenjem i sustave bez kašnjenja.
6. **Topologija** – Ovisno o lokaciji i vezama između agenata postoje statička i dinamička topologija sustava. Statička podrazumijeva da se pozicije i veze

agenata ne mijenjaju tijekom životnog ciklusa agenta dok kod dinamičkih agent može ulaziti, izlaziti iz sustava i pomicati se pa se stoga mijenja njegova pozicija i veze.

7. **Učestalost prijenosa podataka** (engl. *data transmition frequency*) – Dva su načina dijeljenja podataka kod višeagentnih sustava: vremenski potaknuto (engl. *time-triggered*) i potaknuto događajem (engl. *event-triggered*). Kod prvog agent konstanto promatra (engl. *sense*) okolinu te skuplja podatke i nakon definiranog vremenskog intervala ih šalje drugim agentima. Kod drugog agent promatra okolinu samo kada se dogodi određeni događaj te odmah šalje prikupljene podatke drugim agentima.
8. **Mobilnost** – agenti mogu biti statički ili mobilni. Statički su uvijek smješteni na istoj poziciji u okolini dok se mobilni kreću po okolini.

3.2. Usporedba sa sličnim sustavima

Ekspertni sustavi i objekti (objektno orijentirano programiranje) slični su višeagentnim sustavima, no nikako nisu isti. Postoje ključne razlike u ponašanju, komunikaciji i sposobnostima koje će biti ukratko objašnjene kroz ovo poglavlje.

3.2.1. Usporedba s ekspertnim sustavima

Dorri, Jurdak i Kanhere [4] navode nekoliko ključnih razlika između ekspertnih sustava i višeagentnih sustava. Ekspertni sustavi promatraju okolinu i uče o njoj te donose odluke temeljene na prikupljenim informacijama iz okoline i znanjima dok agenti u obzir uzimaju i ciljeve. Ekspertni sustavi mogu komunicirati samo sa unaprijed definiranim entitetima dok agenti unutar višeagentnog sustava mogu komunicirati s bilo kojim agentom. Agenti direktno djeluju na okolinu dok ekspertni sustav savjetuje kontroleru da izvede neku akciju. Kontroler je zasebni sustav koji koristi i druge ulazne informacije te može odbiti odluku ekspertnog sustava.

3.2.2. Usporedba s objektima

Prema Dorriju, Jurdaku i Kanhereu [4], objekt kod objektno orijentiranog programiranja komunicira s limitiranim, unaprijed definiranim objektima putem metoda koje moraju biti javne. Objekt enkapsulira svoje stanje te pruža metode kojima drugi objekti mogu pozivati kako bi provodili akcije nad trenutnim stanjem te ne može kontrolirati učestalost pozivanja tih metoda. S druge strane agenti mogu kontrolirati učestalost zahtjeva za resursima koje dobivaju od drugih agenata te mogu komunicirati s bilo kojim agentom u mreži. Objekti imaju unaprijed definiran i limitiran broj ulaza dok agenti koriste višestruke ulaze.

Schatten [5] navodi kako se agenti razlikuju od objekata po tome što su autonomni, pametni i aktivni. Autonomnost znači da sami odlučuju o tome trebaju li provesti akciju na zahtjev drugog agenta. Pametni su jer su sposobni za kompleksno ponašanje(pro-aktivno, reaktivno i društveno) što objektno orijentirana paradigma ne uzima u obzir. Aktivni su zato jer svaki agent ima barem jednu vlastitu dretvu (engl. *thread*).

4. Pametna mreža

Pametna mreža (engl. *smart grid*) je električna mreža koja koristi ICT tehnologije kako bi omogućila dvosmjernu komunikaciju između korisnika i pružatelja električne energije [6]. Pomoću pametnog brojila (engl. *smart meter*), pametnih kućanskih aparata, obnovljivih izvora energije, pametnih trafostanica i elektrana koji prate događanja u električnoj mreži [7], ali i aktivnim uključivanjem korisnika u rad mreže omogućava smanjenje potrošnje električne energije i troškove, povećava učinkovitost, maksimizira pouzdanost i transparentnost, omogućava kontrolu i optimizaciju obnovljivih izvora energije [6]. Zahvaljujući informacijama u realnom vremenu, omogućava upravljanje potražnjom i gotovo momentalno usklađivanje potražnje i opskrbe na razini individualnih potrošača kao što su zgrade ili čak sami uređaji [6].

2005. godine, Generalna uprava za istraživanje Europske komisije pokrenula je „Europsku tehnološku platformu (ETP) o pametnim mrežama“ (engl. *European SmartGrids Technology Platform*). „ETP o pametnim mrežama“ je, 2006. godine, objavio rad pod nazivom „Vizija i strategija za europske električne mreže u budućnosti“ (engl. Vision and Strategy for Europe's Electricity Netowrks of the Future) [8] u kojem navode kako bi pametne mreže trebale imati sljedeće karakteristike:

3. **Fleksibilnost** (engl. *Flexible*) – Ispunjavanje potreba klijenta te istovremeno reagiranje na promjene i izazove.
4. **Pristupačnost** (engl. *Accessible*) – Odobravanje priključnog pristupa svim korisnicima mreže, osobito za obnovljive izvore energije i visoko učinkovitu lokalnu distribucijsku proizvodnju s nultom ili niskom ugljičnom emisijom.
5. **Pouzdane i otporne** (engl. *Reliable and resilient*) – Osiguravanje i poboljšavanje sigurnosti i kvalitete opskrbe, u skladu sa zahtjevima digitalnog doba s otpornošću na opasnosti i nesigurnosti.
6. **Ekonomične** (engl. *Economic*) – Pružanje najveće vrijednosti kroz inovacije, učinkovito upravljanje energijom i „ravnopravno stanje“ (engl. *level playing field*) natjecanja i regulacije.

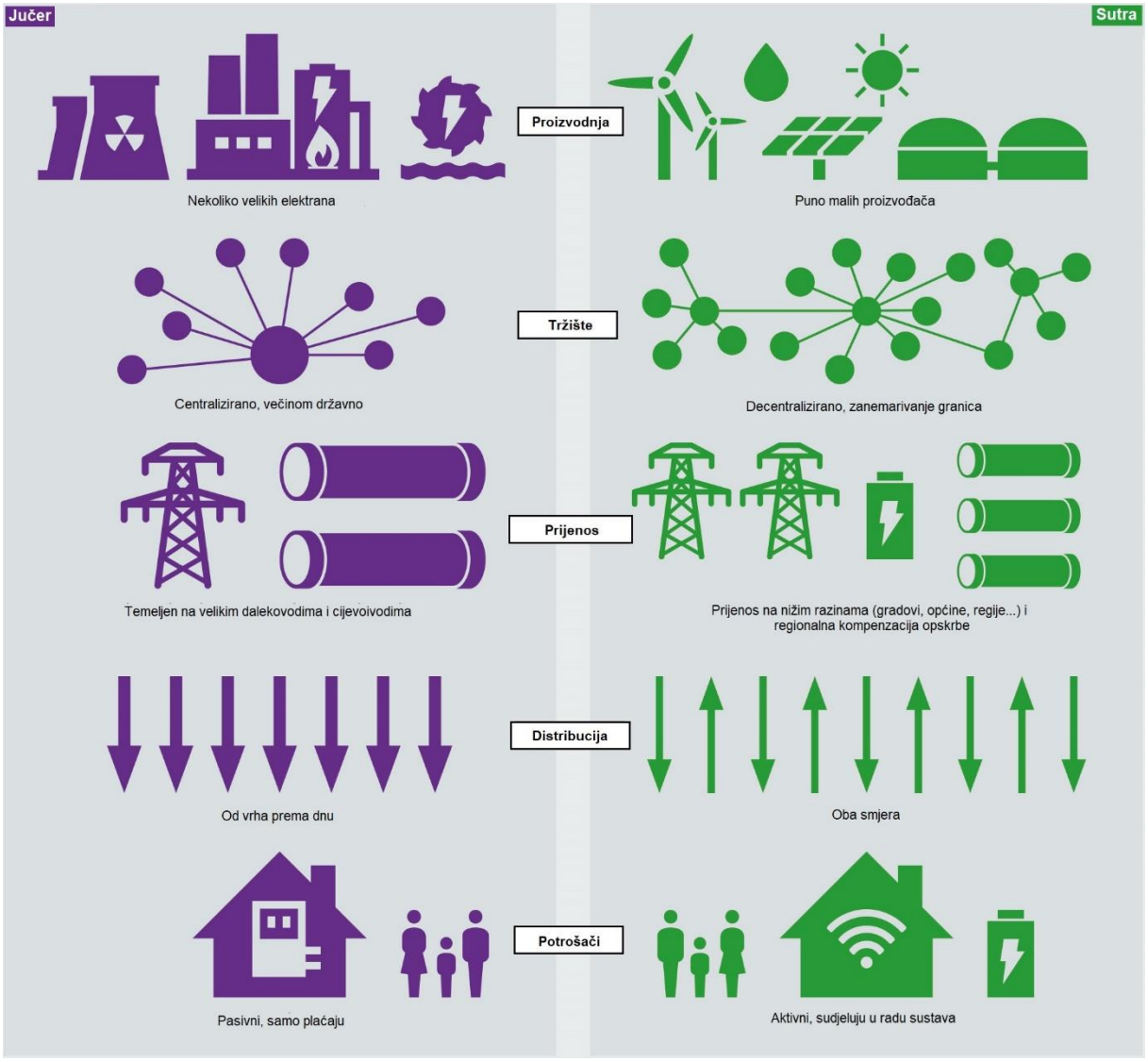
Na Techopedii [9] se kao glavne karakteristike navode:

- **Upravljanje opterećenjem** (engl. *Load handling*) – Opterećenje mreže nije konstantno, stoga bi pametna mreža trebala savjetovati krajnjim korisnicima privremeno smanjenje potrošnje u slučaju prevelikog opterećenja.

- **Podrška za metodu odgovora na potražnju** (engl. *Demand response support*) – Pružanje automatiziranog načina smanjenja računa korisnicima, tako da ih navodi na korištenje uređaja niskog prioriteta samo onda kada je niža cijena struje (zbog niže potražnje).
- **Decentralizacija proizvodnje energije** (engl. *Decentralication of power generation*) – Omogućavanje proizvodnju energije individualnom korisniku, na licu mjesta, primjenom bilo koje odgovarajuće metode po vlastitom nahođenju (npr. Solarni paneli, vjetrenjače…).

Ured isporuke električne energije i energetske pouzdanosti (engl. *Office of Electricity Delivery and Energy Reliability*) američkog odjela za energetiku (engl. *U.S. Department of Energy*) na svojim stranicama [10] posvećenim pametnim mrežama, navodi sljedeće prednosti pametnih mreža:

- Veća učinkovitost prijenosa električne energije
- Brže obnavljanje električne energije nakon poremećaja
- Smanjeni troškovi rada i upravljanja te samim time i niži troškovi električne energije za krajnje korisnike
- Smanjenje vršne potražnje što će pomoći smanjenju stopa električne energije
- Povećana integracija velikih sustava obnovljivih izvora energije
- Bolja integracija klijentskih sustava za proizvodnju električne energije, uključujući i sustave obnovljivih izvora energije
- Poboljšana sigurnost



Slika 15. Karakteristike tradicionalnog sustava (lijevo) i pametne mreže (desno) (Wikipedia [7])

4.1. Pametno brojilo

Pametno brojilo (engl. *smart meter*) je digitalno brojilo koje omogućava dvosmjernu komunikaciju, a time i veću interakciju između elektrane i korisnika, za razliku od tradicionalnog brojila koje pruža informacije samo u jednom smjeru. Pametno brojilo sastoji se od samog brojila koje mjeri potrošnju električne energije, komunikacijskog protokola i kontrolnog uređaja.

Osnovne karakteristike pametnog brojila, prema Morvaju, Lugaricu, Krajcaru [6] su:

- Mjerenje potrošnje električne energije i količine lokalno proizvedene energije u stvarnom vremenu ili približno stvarnom vremenu.
- Može biti očitan na daljinu i lokalno
- Elektrane mogu koristiti brojilo kako bi limitirale količinu energije koja se uzima iz mreže ili šalje u mrežu te mogu čak i potpuno isključiti korisnika.

Pametna brojila komuniciraju s elektranom te elektrana može poslati informaciju o većoj cijeni, zbog povećane potražnje i opterećenja mreže, kako bi savjetovalo korisniku da možda isključi neke uređaje te tako uštedi novce, a samim time i smanji opterećenje mreže. Također, može komunicirati i s kućanskim aparatima te automatizirano upravljati uključivanjem i isključivanjem uređaja ili planiranjem kada će se koji uređaj uključiti te time automatski korisniku uštediti novce. Osim toga, pametna brojila mogu prepoznati energiju koja dolazi iz lokalnih, korisnikovih izvora energije te energiju koja se uzima iz mreže te naplaćivati samo onu koju korisnik uzima iz mreže što će također pridonijeti uštedi kod korisnika.

Pametna brojila mogu komunicirati s uređajima za prijenos i distribuciju kako bi se prilagodile postavke te dobila bolja kontrola nad mrežom, povećala pouzdanost i dostupnost mreže.

5. Odgovor na potražnju

Odgovor na potražnu (engl. *demand response - DR*) je program kojem je cilj poticati korisnike na smanjenje potrošnje u trenutcima vršnog opterećenja mreže ili na povećanje potrošnje kada je opterećenje mreže nisko. To se postiže tako da elektrana šalje signal kada je potrebno prilagoditi opterećenje. Taj signal zaprima centralna jedinica zadužena za automatizaciju zgrade, na primjer pametno brojilo, te isključuje određene uređaje, odgađa početak rada uređaja ili se prebacuje opterećenje na interni izvor električne energije ukoliko je cijena struje viša od zadane. Naravno, cilj je da korisnik određuje postavke temeljem kojih će pametno brojilo odlučiti kada i kako reagirati na ovakve promjene.

Mohagheghi, Stoupis, Wang, Li i Kazemzadeh [11] dijele DR programe u tri grupe, ovisno o strani koja inicira zahtjev za redukcijom:

1. **DR programi bazirani na poticaju** (engl. *Incentive-based DR programs*) – Kod ovog pristupa, pružatelj usluge, na primjer elektrana, šalje signale kojima zahtijeva redukciju prema korisnicima. Ti signali za korisnike mogu biti opcionalni ili obavezni. Primjeri programa koji spadaju pod ovu kategoriju su Izravna kontrola opterećenja (engl. *Direct Load Control* - DLC) i Prekidna i mala opterećenja (engl. *Interruptible & Curtailable Load – I&C*). Kod DLC-a elektrana može direktno upravljati planiranjem pokretanja uređaja ili isključiti uređaj kada je to potrebno. Kod I&C korisnik, ili pametno brojilo u korisničko ime, reagira na signale te prilagođava potrošnju.
2. **DR programi bazirani na tarifi** (engl. *Rate-based DR programs*) – Kod ovih programa cijena električne energije se mijenja bilo u predefiniranom vremenu, na primjer niža cijena tokom noći. Također, može se mijenjati i dinamički ovisno o periodu dana, mjeseca ili godine. Klijeti plaćaju višu cijenu u vremenima vršne potražnje tj. nižu cijenu kada je potražnja smanjena. Cijene se mogu postavljati unaprijed, na dnevnoj bazi, svakog sata ili u stvarnom vremenu. Klijent, ili pametno brojilo u njegovo ime, mogu odlučiti hoće li reagirati na promjenu cijene.
3. **Ponude za smanjenje potražnje** (engl. *Demand reduction bids*) – Kod ovog programa, klijent je taj koji inicira smanjenje potražnje na način da šalje ponudu koja sadrži za koliko može smanjiti potražnju te koja je cijena toga. Ovaj program potiče velike klijente da se izjasne za koju cijenu su voljni smanjiti opterećenje i koliko.

6. Pametna zgrada

Kako navode na stranicama Gemaltoa [12],Pametna zgrada je zgrada koja koristi ICT tehnologije poput bežične komunikacije, senzora i IoT (engl. *Internet of Things*) tehnologija za komunikaciju, prikupljanje i analizu podataka u svrhu upravljanja određenim sustavima unutar zgrade, automatizacije te optimizacije performansi tih sustava. Neki od sustava kojima zgrada upravlja su kontrola pristupa, sigurnost, osvjetljenje, grijanje, ventilacija, klimatizacija, potrošnja električne energije i slično kako bi se povećali komfor, sigurnost i energetska učinkovitost te smanjili troškovi života [12].

Morvaj, Krajcar i Lugaric [6] navode kako se u drugoj polovici sedamdesetih godina prošlog stoljeća, izraz „pametna zgrada“ odnosio na zgrade građene prateći koncept energetske učinkovitosti, dok je u osamdesetima pametna zgrada bila ona kojom se moglo upravljati pomoću osobnog računala.

U današnje vrijeme, navode Morvaj, Krajcar i Lugaric [6], pametna zgrada i dalje koristi koncept iz sedamdesetih i osamdesetih godina 20. stoljeća uz dodatak podsustava za upravljanje obnovljivim izvorima energije, kućanskim aparatima i drugim uređajima te potrošnjom električne energije. Koristeći ICT pametna zgrada komunicira s uređajima unutar same zgrade te njima upravlja, ali komunicira sa svojom okolinom, drugim zgradama, elektranama i slično [6].

Prema Morvaju, Krajcaru i Lugaricu [6], pametna zgrada se općenito sastoji od:

- Senzora – promatraju i detektiraju promjene te o tome šalju poruke obavijesti
- Aktuatora – vrše fizičke akcije
- Kontrolera – kontrolira jedinice i uređaje koristeći pravila koja je postavio korisnik
- Kontrolne jedinice – omogućava programiranje jedinica i uređaja u sustavu
- Sučelja – omogućava komunikaciju korisnika sa sustavom
- Mreže – omogućava komunikaciju između jedinica i uređaja
- Pametnog brojila – omogućava dvosmjernu komunikaciju u realnom ili približno realnom vremenu između korisnika i elektrane

Osim navedenih komponenti, pametna zgrada kojoj je cilj energetska učinkovitost i ušteda električne energije, će uz navedene komponente imati i skladište za električnu energiju te mali obnovljivi izvor električne energije.

Prema Ghent Sveučilištu [13], stambene zgrade i kuće odgovorne su za 40% potrošnje energije i 36% emisije CO_2 u Europskoj uniji, a oko trećina ih je starija od 50 godina. Poboljšanjem energetske učinkovitosti, ukupna potrošnja u Europskoj uniji može se smanjiti 5-6%, a emisija CO_2 oko 5%.

Što se pak tiče same električne energije, zbog vršne potražnje, do 20% ukupnih proizvodnih kapaciteta koristi se samo 5% vremena. Ukoliko se zgrade uključe u *„Demand Response“* program, vršna vrijednost može se smanjiti za 20-50%, a ukupna potrošnja električne energije za 10-15% [14].

Slika 16. Pametna energetska kuća (Morvaj, Krajcar i Lugaric [6])

Slika 16. prikazuje pametnu energetsku kuću koja se sastoji od sljedećih dijelova [6]:

- **Obnovljivi izvor energije** – Pomoću pametnog brojila može se upravljati kako bi se balansirala potražnja i opskrba. Energija koju proizvodi može se prodavati, skladištiti ili koristiti direktno unutar zgrade.

- **Skladište energije** – Može biti električni auto, baterija i slično. Energija se može skladištiti kada je niska potražnja i niska cijena električne energije te koristiti za vrijeme vršne potražnje kada cijena poraste.
- **Pametno brojilo** – Omogućava dvosmjernu komunikaciju i udaljeno čitanje. Korisnik vidi potrošnju i cijenu u realnom vremenu te temeljem tih informacija može programirati kako da pametna zgrada reagira.
- **Pametni uređaji** – Mogu biti dio automatizacije ili funkcionirati zasebno. Komuniciraju s pametnim brojilom te se pale i gase ovisno o uvjetima koje je korisnik postavio.
- **Automatizacija** – Sastoji se od senzora, aktuatora, kontrolera, centralne jedinice, sučelja i mrežnog standarda za komunikaciju. Omogućava korisniku da programira ponašanje zgrade temeljem uvjeta.
- **Širokopojasna veza** – Omogućava komunikaciju s mrežom i drugim pametnim zgradama koristeći komunikacijske standarde te tako kreira aktivne mikro-mreže (engl. *microgrids*)

7. Implementacija

Pametna kuća u ovom radu bit će fokusirana na uštedu električne energije i smanjenje troškova prilikom korištenja električne energije. Kuća će biti dio pametne mreže te će reagirati na promjene cijene električne energije i odlučivati o tome kada pojedini uređaj može početi raditi te iz kojeg izvora će se trošiti električna energija. Bit će moguće vidjeti koja je trenutna cijena i potrošnja električne energije te kolika je predviđena proizvodnja električne energije putem solarnih panela u sljedećih 48 sati.

Kako bi se postiglo opisano ponašanje, uređaji poput kućanskih aparata, solarnih panela i pametnog brojila bit će implementirani kao agenti koji će tvoriti višeagentni sustav. Za programiranje agenata, njihova ponašanja i komunikaciju koristit će se Java Agent DEvelopment Framework (skraćeno JADE), a neki će podaci biti dohvaćani i putem REST web servisa.

7.1. Korištene tehnologije

Kao što je navedeno u uvodu ovog poglavlja, za implementaciju agenata, njihova ponašanja i komunikaciju korišten je JADE programski okvir te će taj programski okvir biti detaljnije razrađen u ovom poglavlju.

Također, korišteni su i Googleov geocoding REST web servis za dohvaćanje geografske širine i dužine prema adresi te Forecast.Solar REST web servis za dohvaćanje predviđanja koliko će električne energije proizvesti solarni paneli. Pozivanje je prikazano u poglavlju 7.4.4, a izgled odgovora prikazanje u poglavlju 7.5.2.

7.1.1. Java Agent DEvelopment Framework

Java Agent DEvelopment Framework (kraće JADE) je *open source* programski okvir potpuno implementiran u Java programskom jeziku koji služi za razvoj aplikacija baziranih na agentima. JADE olakšava i pojednostavljuje implementaciju višeagentnih sustava kroz *midleware* koji je u skladu s FIPA (Foundation for Intelligent Physical Agents) specifikacijama. Sustavi bazirani na JADE platformi mogu biti distribuirani na različitim

uređajima i različitim operativnim sustavima, a konfiguracija se može kontrolirati putem udaljenog grafičkog sučelja. [15]

JADE platforma omogućava *peer-to-peer* komunikaciju asinkronim porukama, implementaciju FIPA-ACLa (*FIPA Agent Communication Language*), jednostavno i moćno sastavljanje i izvršavanje zadatak, objavljivanje i pronalazak servisa pomoću žutih stranica te mnogo drugih funkcionalnosti važnih za razvoj distribuiranih sustava. [15]

JADE se sastoji od [16]:

- **Izvršnog okruženja** (engl. *runtime environment*) u kojem agenti žive i koje mora biti pokrenuto na hostu prije nego što na njemu želimo pokrenuti agenta.
- **Biblioteke** (engl. *library*) koja pruža klase potrebne za razvoj agenata.
- **Grafičkih alata** za monitoring i administraciju pokrenutih agenata.

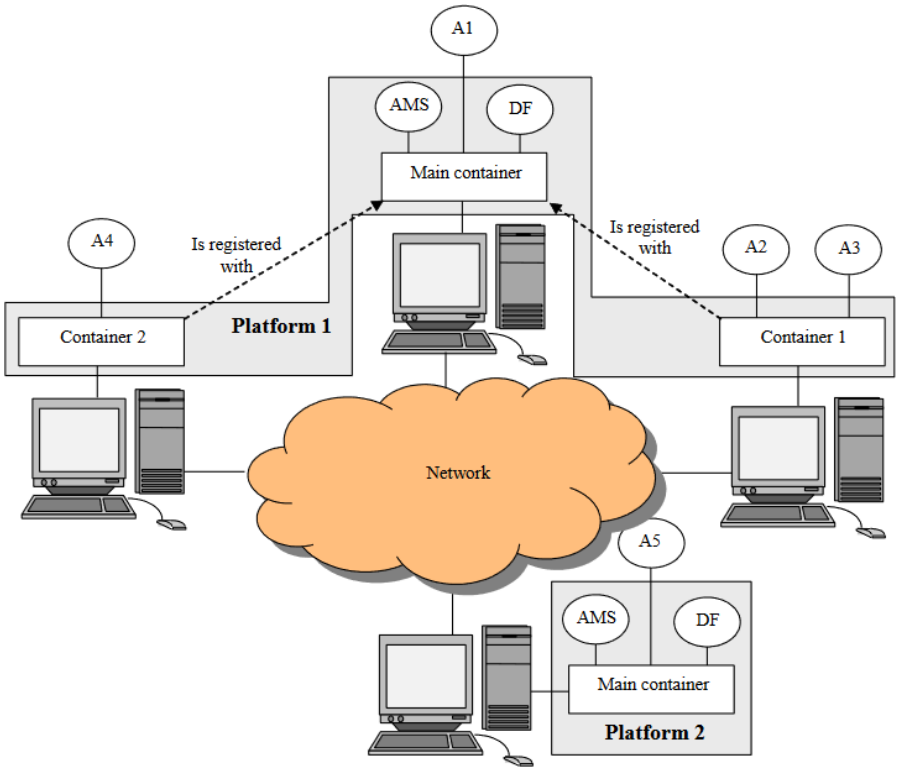
Svaka instanca izvršnog okruženja naziva se kontejner (engl. *Container*). Kontejner može biti glavni (engl. *Main container*) i „normalni“ te može sadržavati više agenata. Više aktivnih kontejnera čini platformu (engl. *Platform*). Svaka platforma može imati samo jedan glavni kontejner te više „normalnih“ kontejnera koji se registriraju na glavni kontejner tako da im se kaže na kojem hostu (IP adresa ili URL) i portu se on nalazi. [16]

Glavni kontejner se razlikuje od normalnog kontejnera po tome što sadrži dva posebna agenta, AMS (*Agent Management System*) i DF (*Directory Facilitator*), koji se pokreću automatski prilikom pokretanja glavnog kontejnera. [16]

AMS predstavlja autoritet unutar platforme [16] te samo on može kreirati i uništiti agente, uništiti kontejnere i zaustaviti platformu [17]. Svaki agent mora se registrirati kod AMS-a kako bi dobio jedinstveni identifikator (engl. *Agent identificator - AID*) [18]. AMS pruža usluge životnog ciklusa (engl. *life-cycle*) i bijelih stranica (engl. *white-pages*), održavanja direktorija identifikatora i stanja agenta [17]. Samo jedan AMS može postojati unutar platforme [18].

DF pruža uslugu žutih stranica kako bi agent pronašao drugog agenta koji mu može pružiti usluge potrebne da bi postigao svoje ciljeve [16].

Slika 17. pokazuje gore opisani koncept. *Main Container* te *Container 1* i *Container 2* čine platformu nazvanu *Platform 1*. *Container 1* i *Container 2* su registrirani kod *Main Containera*. *Main Container* sadrži AMS, DF i A1 agente, *Container 1* sadrži A2 i A3 agente, a *Container 2* sadrži agenta A5. Platforma *Platform 2* sastoji se samo od *Main Containera* koji sadrži AMS, DF i A5 agente.



Slika 17. JADE koncept [16]

7.1.1.1. Klasa Agent

Klasa `Agent` je osnovna klasa koju je potrebno naslijediti prilikom definiranja vlastitih agenata. Ona pruža funkcionalnosti za ostvarivanje osnovne interakcije s platformom, poput registracije, konfiguracije, upravljanja na daljinu i slično, te osnovni set metoda koje omogućavaju implementaciju ponašanja agenta.

Neke od metoda koje klasa `Agent` sadrži, ali su prazne i potrebno ih je implementirati su `setup()` i `takeDown()`. Metoda `setup()` je namijenjena da se u nju doda kod za inicijalizaciju agenta, a poziva se nakon što je agent postavljen u aktivno (engl. *Active*) stanje. Metoda `takeDown()` se poziva kada se agent prebacuje u obrisano (engl. *Deleted*) stanje te služi za implementiranje koda za „čišćenje“ (engl. *cleanup*) i odjavljivanje (engl. *deregistering*) agenta sa DF agenta - `DFService.deregister(this)`. Tu su još i metode koje se pozivaju prilikom premještanja agenta, `beforeMove()` i `afterMove()` te prilikom kloniranja agenta, `beforeMove()` i `afterMove()`.

Naravno, postoje i metode koje su već implementirane, a najznačajnije su one za slanje poruka, `send(ACLMessage msg)`, primanje poruka, `receive()` i

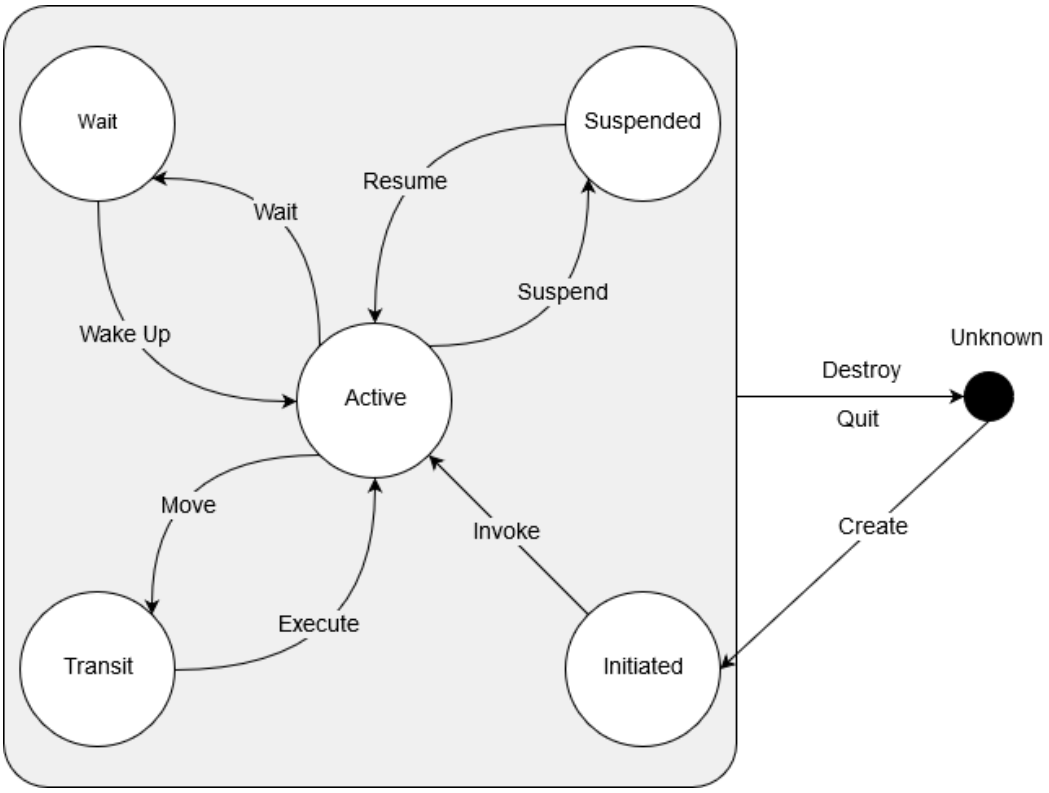
`blockingReceive()` te za dodavanje i brisanje ponaša, `addBehaviour(Behaviour b)`, `removeBehaviour()`.

Klasa `Agent` ima i metode za prelazak iz jednog stanja u drugo čiji format je `doXXX()`, gdje XXXX predstavlja ime stanja, na primjer `doSuspend()`. Priručnik za programere [18] navodi da su stanja sukladna FIPA specifikaciji životnog ciklusa agenata (slika 18.) te objašnjava sljedeća dostupna stanja:

- **Inicirano** (engl. *Initiated*) – Agentni objekt je kreiran, ali nije registriran kod AMS agenta, nema ime ni adresu te ne može komunicirati s drugim agentima.
- **Aktivno** (engl. *Active*) – Agentni objekt je registriran kod AMS agenta, ima ime i adresu te moože koristiti različite JADE funkcionalnosti.
- **Suspendirano** (engl. *Suspended*) – Agent je trenutno zaustavljen, njegova interna dretva je suspendirana i ne izvršava se ni jedno ponašanje.
- **Čekanje** (engl. *Waiting*) – Agent je blokiran, interna dretva mu je u stanju „spavanja“ (engl. *sleeping*) te će se „probuditi“ kada se zadovolji neki uvjete (najčešće kada pristigne poruka).
- **Obrisano** (engl. *Deleted*) – agent je definitivno „mrtav“ (engl. *dead*), dretva mu je prekinuta i više nije registriran kod AMS agenta.
- **Tranzitno** (engl. *Transit*) – mobilni agent prelazi u ovo stanje kada seli na drugu lokaciju. Sustav i dalje zaprima poruke u međuspremnik te će ih dostaviti na novu adresu.

Svaki agent mora imati i jedinstveni identifikator, AID, koji mu dodjeljuje AMS prilikom kreiranja, kroz konstruktor. AID je, također, u skladu s FIPA specifikacijom te se sastoji od tri dijela [18]:

1. Globalno jedinstvenog imena koje JADE, zadano, sastavlja od lokalnog imena, tj. imena agenta proslijeđenog kod kreiranja agenta, znaka @ te identifikatora platforme koji se, zadano sastavlja kao **<ime računala glavnog kontejnera>** „**:**“ **<port glavnog kontejnera>/JADE**, ali se može i eksplicitno definirati. Primjer imena je: *Agent1@172.21.224.1:12345/JADE*.
2. Skupa transportnih adresa koje nasljeđuje od platforme na kojoj je pokrenut.
3. Skupa rezolvera (engl. *resolver*), to jest „*white page*“ servisa kod kojih je registriran.



Slika 18. Životni ciklus agenta prema definiciji FIPAe [18]

7.1.1.2. Klasa Behaviour

Klasa `Behaviour` je apstraktna klasa koju nasljeđuju druge `Behaviour` klase pomoću kojih se definira logika, to jest zadaci koje agent mora obavljati. Programsku logiku potrebno je smjestiti unutar metode `action()`. Agentu se može dodati više ponašanja te se ta ponašanja izvršavaju istovremeno. Unutar `setup()` metode agenta potrebno je dodati barem jedno ponašanje pomoću agentove metode `addBehaviour(Behaviour b)`. Čim `setup()` metoda završi s izvršavanjem zakazuje se i izvršavanje ponašanja. Automatski se pokreće prvo ponašanje iz reda čekanja te se prelazi na sljedeće ponašanje. Ponašanja se pokreću križno s time da se `action()` metoda nikada ne prekida kako bi se pokrenulo drugo ponašanje, odnosno, ako se izvršava `action()` metoda nekog ponašanja, niti jedno drugo ponašanje me može započeti. Takav način rada na Engleskom jeziku naziva se *round-robin non-preemptive way*. Za upravljanje rasporedom pokretanja ponašanja zadužen je objekt klase `Scheduler` unutar objekta klase `Agent`.

JADE pruža niz klasa koje direktno ili indirektno nasljeđuju `Behaviour` klasu te, uglavnom, nije potrebno kreirati vlastite klase ponašanja koje će direktno nasljeđivati ovu

apstraktnu klasu. Obzirom da postoji podosta `Behaviour` podklasa, bit će navedene samo neke od njih:

- **SimpleBehaviour** – Direktna podklasa klase `Behaviour` koja služi za implementaciju jednostavnih ponašanja
  - **OneShotBehaviour** – podklasa klase `SimpleBehaviour`. Ponašanje koje će se izvršiti samo jednom.
  - **CyclicBehaviour** - podklasa klase `SimpleBehaviour`. Izvršavat će se kružno dok god je agent „živ“.
  - **TickerBehaviour** – Slično kao i `CyclicBehaviour` samo što se izvršava u određenim vremenskim intervalima.
- **CompositeBehaviour** – Direktna podklasa klase `Behaviour` koja služi za implementaciju složenih ponašanja
  - **ParallelBehaviour** – Direktna podklasa klase `CompositeBehaviour`. Dodaju mu se ponašanja koja će ovo ponašanje kontrolirati i koja se izvršavaju paralelno. Uvjet za kraj rada ovog ponašanja moguće je definirati. Paralelno ponašanje može završiti kada su sva dodana ponašanja završila, kada je N dodanih ponašanja završilo ili kada je jedno od dodanih ponašanja završilo**.**
  - **SquentialBehaviour** – Nasljeđuje `SerialBehaviour` klasu koja je apstraktna i koja nasljeđuje `CompositeBehaviour`. Dodaju mu se ponašanja koja se izvršavaju jedno za drugim te završava s izvršavanjem kada se završi zadnje dodano ponašanje.

Valja zapamtiti kako se ponašanja dodana direktno na agenta pomoću `addBehaviour` metode izvršavaju paralelno.

7.1.1.3. Interakcijski protokoli

Interakcijski protokoli služe za izgradnju kompleksnije komunikacije između agenata, a definirani su od strane FIPAe. JADE sadrži klase za implementaciju interakcijskih protokola koje slijede FIPA definicije, indirektno nasljeđuju `Behaviour` klasu, a nalaze su u paketu `jade.proto`. Za svaki interakcijski protokol JADE pruža dvije klase, `Initiator` klasu i `Responder` klasu. Ove klase direktno nasljeđuju `FMSBehavior` (konačni automat, engl.

Finte State Machine - FMS) klasu koja nasljeđuje `SerialBehaviour` klasu. Jedine iznimke su `SimpleAchieveREInitiator` i `SimpleAchieveREResponder` klase koje nasljeđuju `SimpleBehaviour`.

`Initiator` ponašanja služe za pokretanje komunikacije. Čim dosegnu konačno stanje, prekidaju se i miču iz agentovog reda čekanja zadataka, to jest iz *scheduler* objekta agenta. Ukoliko se želi ponovno iskoristiti isti `Initiator` objekt, moguće je pozvati neku od `reset` metoda koje `Initiator` sadrži te ga je potrebno dodati na agenta pomoći `addBehaviour` metode. `Initiator` može poslati poruku na više primatelja te čeka odgovore svih primatelja ukoliko nije definiran rok u kojem odgovor treba pristići.

`Responder` ponašanja služe za odgovaranje kada je agent kontaktiran od strane drugog agenta. Za razliku od `Initiatora`, `Responder` ponašanja su kružna te se ponovno planira njihovo izvođenje čim dosegnu bilo koje završno stanje. Jedina iznimka `Responder` klase s jednom sesijom čija imena započinju sa SS (engl. *single session - SS*).

U ovom radu bit će korišteni `ProposeInitiator` (7.4.3) i `ProposeResponder` (7.4.2) koji implementiraju FIPA-propose protokol za slanje prijedloga i odgovor na isti te `SubscriptionInitiator` (7.4.2) i `SubscriptionResponder` (7.4.1) koji implementiraju FIPA-subscription protokol za pretplatu i dobivanje notifikacija od agenta na kojeg se agent pretplatio.

Osim gore navedenih klasa JADE sadrži još i sljedeće klase:

- **AchieveREInitiator/Responder**, **SimpleAchieveREInitiator/Responder, IteratedAchieveREInitiator/Responder** te **SSIteratedAchieveREResponder** – implementiraju FIPA-Request, FIPA-query, FIPA-Request-When, FIPA-recruiting, FIPA-brokering .
- **ContractNetInitiator/Responder** i **SSContractNetResponder** – implementiraju FIPA-Contract-Net protokol.

7.1.1.4. Komunikacija Agenta i ACLMessage klasa

Komunikacija između agenata je jedna od najbitnijih funkcionalnosti koje pruža JADE [16]. Svaki agent ima red čekanja u koji se spremaju poruke poslane od drugih agenata. Čim je poruka postavljena u red čekanja, agent je o tome obaviješten, a programer je taj koji određuje kada će agent uzeti i procesirati tu poruku. Poruke se prenose asinkrono.

U poglavlju 7.1.1.1 navedeno je da klasa `Agent` pruža metode za slanje poruka, `send(ACLMessage msg)` i primanje poruka, `receive()` i `blockingReceive()`. Valja napomenuti da su metode za primanje poruka preopterećene pa im se može proslijediti parametre poput instance `MessageTemplate` klase ili vrijeme u milisekundama koliko agent treba čekati na poruku. Kod korištenja metoda za primanje poruka treba znati da `blockingReceive` metode blokiraju sve aktivnosti i sva ponašanja agenta do zaprimanja poruke ili do isteka zadanog vremena, dok `receive` metode odmah vraćaju `null` ukoliko poruka nije prisutna u redu čekanja (engl. *queue*).

Poruke su u JADEu predstavljene klasom `ACLMessage` koja slijedi FIPA specifikaciju jezika za komunikaciju agenata (engl. *Agent Communication Language - ACL*).

Neka od polja koja sadrži ova klasa su:

- Lista primatelja (engl. *receivers*) u koju se dodaju AIDovi agenata kojima se želi poslati poruka
- Pošiljatelj (engl. *sender*) u kojem je spremljen AID agenta koji šalje poruku
- Namjera komunikacije, engleskog naziva *performative* koje nosi informaciju o tome što pošiljatelj želi postići slanjem te poruke. Moguće vrijednosti određene su konstantama unutar same klase
- Sadržaj (engl. *content*)
- Jezik (engl. *language*) koje služi za definiranje jezika koji je korišten u sadržaju
- Ontologija (engl. *ontology*) koje služi za specificiranje onotlogije korištene u sadržaju kako bi se postigla interoperabilnost s agentima pisanim u drugim programskim jezicima
- Identifikator razgovora (engl. *ConversationID*) koje služi za povezivanje poruka iz istog razgovora.

Komunikaciju je moguće implementirati na tri načina [19]:

1. Koristeći niz znakova (engl. *string*) za sadržaj poruke, što je u redu ukoliko se želi prenijeti neka jednostavna poruka, ali ne i kada sadržaj poruke treba biti neki apstraktni koncept, objekt ili strukturirani podatak.
2. Koristeći serijalizaciju za slanje Java objekata kao sadržaja poruke, što je uredu ukoliko su svi agenti napisani u Java programskom jeziku.

3. Definirajući ontologiju za objekte koji će se slati kako bi se poruke mogle kodirati i dekodirati u standardni FIPA format. Ovaj pristup omogućava komunikaciju s drugim agentnim sustavima, koji mogu biti napisani u drugim programskim jezicima. O samoj ontologiji bit će više riječi u poglavlju 7.3.

7.2. Opis sustava

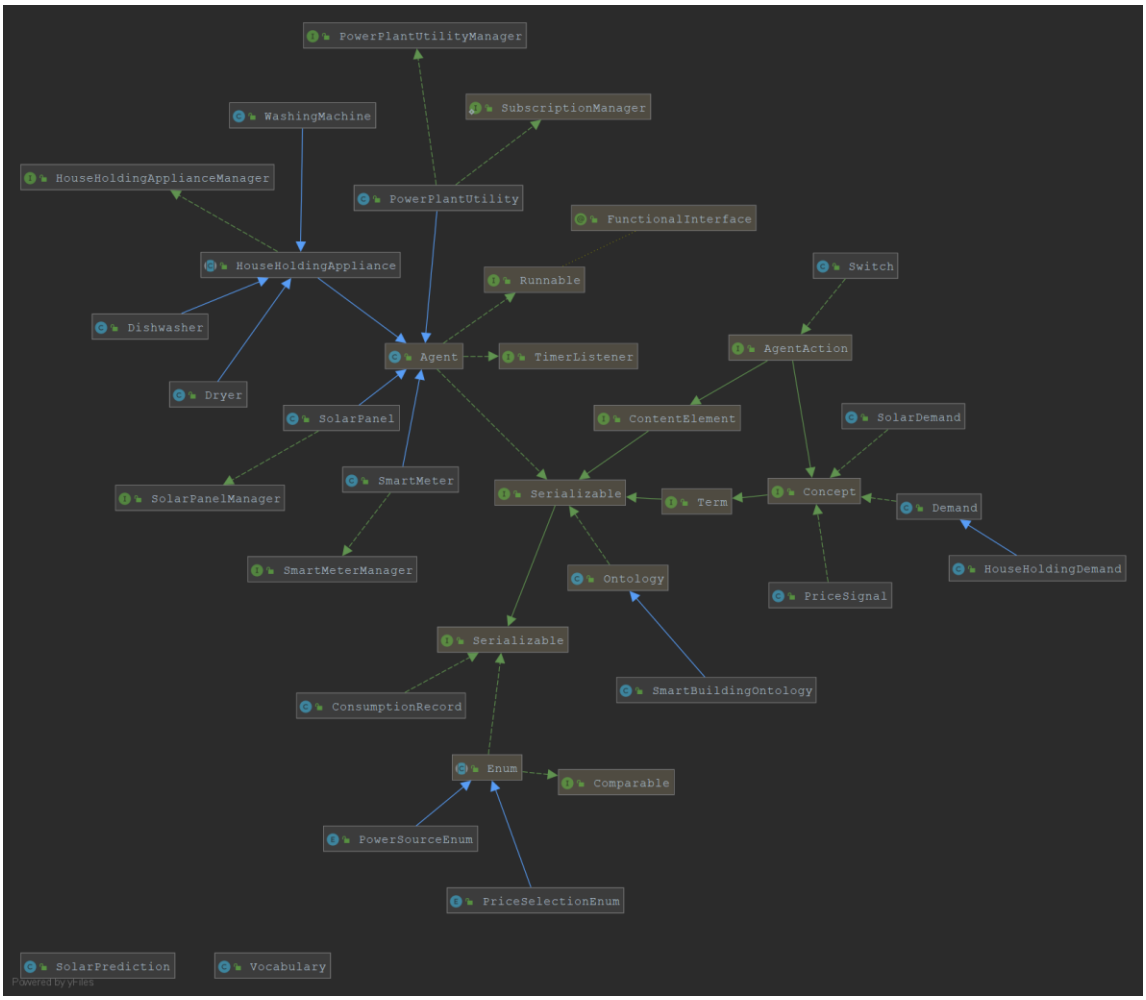
Višeagentni sustav pametne zgrade sastoji se od agenata kućanskih aparata, agenta pametnog brojila, agenta solarnog panela i agenta elektrane. Kućanski aparati predstavljeni su klasama `Dryer`, `Dishwasher` i `WashingMachine` te svi nasljeđuju klasu `HouseHoldingAppliance`. Pametno brojilo predstavljeno je klasom `SmartMeter`, solarni panel klasom `SolarPanel`, a elektrana klasom `PowerPlantUtility`.

Prilikom pokretanja pametnog brojila, pametno brojilo šalje zahtjev za pretplatom (engl. *subscription*) elektrani. Elektrana prihvaća pretplatu te šalje informacije o cijenama i kapacitetu elektrane te obavještava pametno brojilo prilikom promjena u cijenama. Kada se neki od pametnih uređaja želi pokrenuti, šalje zahtjev pametnom brojilo koje, temeljem trenutne cijene i postavke korisnika, određuje može li se uređaj pokrenuti ili ga dodaje u red čekanja te će ga obavijestiti o pokretanju kada cijena padne. Pametno brojilo može prebaciti opskrbu energije sa elektrane na obnovljivi izvor, ukoliko mu agent solarnog panela odgovori kako može zadovoljiti trenutnu potražnju.

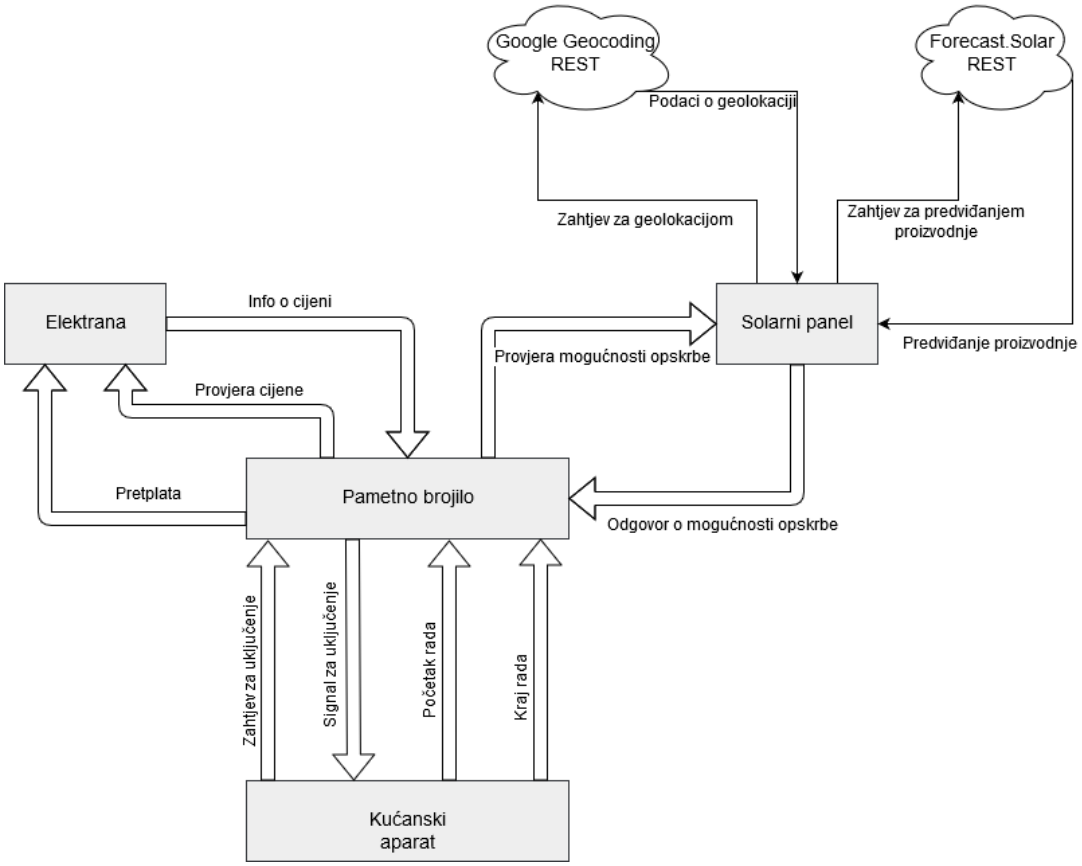
Ovakav sustav, koji reagira na promjene u cijeni te temeljem toga prilagođava potrošnju je I&C program koji spada pod DR programe bazirane na poticaju spomenute u poglavlju 5.

Kako bi se osigurala mogućnost komunikacije s drugim sustavima, napisanim u drugim programskim jezicima, kreirana je i vlastita ontologija.

Slika 19. prikazuje dijagram klasa implementiranog sustava, a slika 20. prikazuje komunikaciju između agenata unutar sustava.



Slika 19. Dijagram klasa



Slika 20. Komunikacija u sustavu

7.3. Ontologija

Ontologija se sastoji od tri dijela, klasa koje će predstavljati sadržaj poruke, `Vocabulary` klase i `SmartBuildingOntology` klase.

Klase koje predstavljaju sadržaj poruke su obične POJO (engl. *Plain Object Java Object*) klase koje sadrže određena polja te `getter` i `setter` metode. Te klase moraju implementirati jedno od sljedećih sučelja:

- **AgentAction** – koristi se kada jedan agent zahtijeva od drugoga da poduzme neku radnju
- **Predicate** – koristi se kada jedan agent pita drugog je li neka tvrdnja točna
- **Concept** – koristi se za poruke koje nisu niti akcija niti ispitivanje tvrdnje.

U definiranju ontologije ovog sustava, klase koje predstavljaju sadržaj poruka implementiraju ili `AgentAction` ili `Concept` sučelje dok `Predicate` nije bio potreban.

7.3.1. Klasa Demand

`Demand` klasa predstavlja sadržaj poruke koja će se slati elektrani kada se želi pokrenuti novi uređaj kako bi se provjerilo hoće li se dogoditi promjena cijene. Ova klasa implementira sučelje `Concept`

```
package ontology;

import jade.content.Concept;

import java.util.Calendar;
import java.util.Date;

public class Demand implements Concept {

    private Integer power;

    // getter and setter
}
```

7.3.2. Klasa HouseHoldingDemand

`HouseHoldingDemand` klasa nasljeđuje `Demand` klasu te predstavlja sadržaj poruke koji će kućanski aparati slati pametnom brojilu kada se žele pokrenuti.

```
package ontology;

import java.util.Date;

public class HouseHoldingDemand extends Demand {

    private Integer duration;

    private Date startDate;

    // getters and setters
}
```

7.3.3. Klasa SolarDemand

`SolarDemand` klasa predstavlja sadržaj poruke koji će pametno brojilo slati solarnom panelu kako bi provjerilo ima li dovoljno kapaciteta da se opskrba električnom energijom prebaci na njega. Ova klasa implementira sučelje `Concept`.

```
package ontology;

import jade.content.Concept;
import jade.util.leap.List;
```

```
public class SolarDemand implements Concept {

    private List demandList;

    // getter and setter
}
```

7.3.4. Klasa PriceSignal

`PriceSignal` klasa predstavlja sadržaj poruke koji će elektrana slati pametnom brojilu kada dođe do promjene cijene ili raspona cijena. Ova klasa implementira sučelje `Concept`.

```
package ontology;

import jade.content.Concept;

public class PriceSignal implements Concept {

    private Float currentPrice;

    private Float minPrice;

    private Float maxPrice;

    // getters and setters
}
```

7.3.5. Klasa Switch

`Switch` klasa predstavlja poruku koju će pametno brojilo poslati kućanskom aparatu kada kućanski aparat može započeti s radom. Obzirom da se kod ove poruke zahtijeva akcija, ova klasa implementira sučelje `AgentAction`.

```
package ontology;

import jade.content.AgentAction;

public class Switch implements AgentAction {

    private boolean action;

    // getter and setter
}
```

7.3.6. Klasa Vocabulary

Klasa `Vocabulary` opisuje terminologiju koncepata koji će se koristiti kao sadržaj poruke. Ona sadrži konstante s nazivima klasa koje predstavljaju sadržaje poruka te nazivima polja koja te klase sadrže.

Isječak koda koji slijedi prikazuje samo dio `Vocabulary` klase koji sadrži konstante vezane uz `PriceSignal` klasu.

```
package ontology;

public class Vocabulary {

    public static final String PRICE_SIGNAL = "PriceSignal";

    public static final String PRICE_SIGNAL_CURRENT_PRICE = "currentPrice";

    public static final String PRICE_SIGNAL_MIN_PRICE = "minPrice";

    public static final String PRICE_SIGNAL_MAX_PRICE = "maxPrice";

    . . .

}
```

7.3.7. Klasa SmartBuildingOntology

Klasa `SmartBuildingOntology` nasljeđuje klasu `Ontology` te služi za definiranje ontologije. Ona opisuje nomenklaturu veza između koncepata koji će se koristiti kao sadržaj poruke njihovu semantiku i strukturu.

Sljedeći isječak koda prikazuje definiranje sheme koncepta klase `SolarDemand` i `HouseHoldingDemand`. Sheme koncepata ostalih klasa su definirane na isti način.

```
package ontology;

import jade.content.onto.BasicOntology;
import jade.content.onto.Ontology;
import jade.content.onto.OntologyException;
import jade.content.schema.AgentActionSchema;
import jade.content.schema.ConceptSchema;
import jade.content.schema.ObjectSchema;
import jade.content.schema.PrimitiveSchema;

import static ontology.Vocabulary.*;


public class SmartBuildingOntology extends Ontology {

    public static final String ONTOLOGY_NAME = "Smart-Building-Ontology";
```

```
    private static Ontology instance = new SmartBuildingOntology();

    public static Ontology getInstance() {
        return instance;
    }

    private SmartBuildingOntology() {
        super(ONTOLOGY_NAME, BasicOntology.getInstance());

        try {

            // HouseHoldingDemand
            cs = new ConceptSchema(HOUSE_HOLDING_DEMAND);
            add(cs, HouseHoldingDemand.class);
            cs.add(DEMAND_POWER, (PrimitiveSchema)
getSchema(BasicOntology.INTEGER), ObjectSchema.MANDATORY);
            cs.add(HOUSE_HOLDING_DEMAND_DURATION, (PrimitiveSchema)
getSchema(BasicOntology.INTEGER), ObjectSchema.MANDATORY);
            cs.add(HOUSE_HOLDING_START_DATE, (PrimitiveSchema)
getSchema(BasicOntology.DATE), ObjectSchema.OPTIONAL);

            // Price Signal
            cs = new ConceptSchema(PRICE_SIGNAL);
            add(cs, PriceSignal.class);
            cs.add(PRICE_SIGNAL_CURRENT_PRICE, (PrimitiveSchema)
getSchema(BasicOntology.FLOAT), ObjectSchema.MANDATORY);
            cs.add(PRICE_SIGNAL_MIN_PRICE, (PrimitiveSchema)
getSchema(BasicOntology.FLOAT), ObjectSchema.MANDATORY);
            cs.add(PRICE_SIGNAL_MAX_PRICE, (PrimitiveSchema)
getSchema(BasicOntology.FLOAT), ObjectSchema.MANDATORY);

            . . .

        } catch (OntologyException oe) {
            oe.printStackTrace();
        }
    }
}
```

7.4. Struktura agenata

Kao što je ranije navedeno agenti nasljeđuju klasu `Agent` te su im dodana ponašanja. Svaki agent ima barem jedno ponašanje. Klase agenata sadrže polja, metode i unutarnje klase koje ih strukturiraju i koje im omogućavaju izvršavanje zadataka.

7.4.1. Agent PowerPlantUtility

Klasa `PowerPlantUtility` predstavlja agenta elektrane. Agent elektrane omogućava agentima pametnog brojila da se pretplate na njega te ih obavještava o svakoj promjeni cijene. Osim toga, pametna brojila mogu poslati poruku kako bi provjerili hoće li se promijeniti cijena ukoliko dozvole uključenje nekog uređaja.

Ova klasa, osim što nasljeđuje klasu agent, implementira i sučelja `PowerPlantUtilityManager` te `SubscriptionManager`.

Sučelje `PowerPlantUtilityManager` služi kako bi se mogle vršiti radnje nad agentom „iz vana“, u slučaju ovog projekta, iz `Main` klase koja će služiti za pokretanje simulacije. Sadrži metode za postavljanje minimalne i maksimalne cijene, trenutne cijene i trenutne potražnje te za dohvaćanje trenutne potražnje.

```
void setMinPrice(BigDecimal price);

void setMaxPrice(BigDecimal price);

void setMinAndMaxPrice(BigDecimal minPrice, BigDecimal maxPrice);

void setCurrentPowerConsumption(Integer currentPowerConsumption);

Integer getCurrentPowerConsumption();

void setCurrentPrice(BigDecimal currentPrice);
```

Postavljanje trenutne potrošnje ili promjena minimalne ili maksimalne cijene, automatski pokreće kalkulaciju trenutne cijene te obavještava pretplaćene agente.

Sučelje `SubscriptionManager` sadrži dvije metode koje se pozivaju kada pametno brojilo zatraži pretplatu ili kada želi odjaviti pretplatu.

```
boolean register(SubscriptionResponder.Subscription var1) throws
RefuseException, NotUnderstoodException;

boolean deregister(SubscriptionResponder.Subscription var1) throws
FailureException;
```

Klasa nadjačava `setup` metodu iz klase `Agent` te u njoj registrira `PowerPlantUtilityManager` sučelje, ontologiju i jezik.

```
super.setup();

registerO2AInterface(PowerPlantUtilityManager.class, this);
getContentManager().registerLanguage(codec);
getContentManager().registerOntology(ontology);
```

Nakon toga poziva metodu `register` u kojoj se čitaju ulazni parametri koji se prosljeđuju kod pokretanja agenta te se kreiraju `DFAgentDescription` i `ServiceDescription` te se agent registrira kod DF agenta. Ulazni parametri koje je moguće proslijediti su minimalna cijena (`minprice`) maksimalna moguća cijena (`maxprice`) te maksimalna snaga elektrane (`maxpower`).

```
private void register() {
    final DFAgentDescription dfd = new DFAgentDescription();
    dfd.setName(getAID());

    for (final Object arg : getArguments()) {
        final String[] splitArg = arg.toString().split(":");
        final String argType = splitArg[0].toLowerCase();
        final String argValue = splitArg[1];
        switch (argType) {
            case "maxprice":
                priceSignal.setMaxPrice(new
BigDecimal(argValue).floatValue());
                break;
            case "minprice":
                final float minPrice = new
BigDecimal(argValue).floatValue();
                priceSignal.setMinPrice(minPrice);
                priceSignal.setCurrentPrice(minPrice);
                break;
            case "maxpower":
                maxPowerOutput = Integer.valueOf(argValue);
                break;
            default:
                break;
        }
    }

    final ServiceDescription sd1 = new ServiceDescription();
    sd1.setType("PowerPlantUtility");
    sd1.setName(getLocalName());
    dfd.addServices(sd1);

    try {
        DFService.register(this, dfd);
    } catch (final FIPAException fe) {
        Logger.println(fe.toString());
    }
}
```

Nakon `register` metode poziva se metoda `calculatePriceStep` koja računa za koliko se povećava cijena ako se potražnja poveća za 1 W.

```
private void calculatePriceStep() {

    final BigDecimal maxPrice =
BigDecimal.valueOf(priceSignal.getMaxPrice().doubleValue()).setScale(2,
RoundingMode.HALF_UP);
    final BigDecimal minPrice =
```

```
BigDecimal.valueOf(priceSignal.getMinPrice().doubleValue()).setScale(2,
RoundingMode.HALF_UP);

    priceStep =
maxPrice.subtract(minPrice).divide(BigDecimal.valueOf(maxPowerOutput), 5,
RoundingMode.HALF_UP);
```

Zatim se agentu dodaju ponašanja.

Prvo dodano ponašanje `PriceSubscriptionResponder` koje nasljeđuje `SubscriptionResponder` klasu. Ono prima zahtjev za pretplatom od pametnog brojila, dodaje ga na listu pretplatnika putem `register` metode iz `SubscriptionManager` sučelja te odmah odgovara brojilu s informacijama o cijeni šaljući mu kao sadržaj poruke objekt klase `PriceSignal`. Sve se to odvija u metodi `handleSubscription`.

```
@Override
protected ACLMessage handleSubscription(ACLMessage subscription) throws
NotUnderstoodException, RefuseException {
    super.handleSubscription(subscription);

    final ACLMessage reply = subscription.createReply();

    try {

        reply.setPerformative(ACLMessage.INFORM);
        getContentManager().fillContent(reply, new
Action(subscription.getSender(), priceSignal));

    } catch (Codec.CodecException | OntologyException e) {
        Logger.println(e.toString());
        reply.setPerformative(ACLMessage.FAILURE);
    }

    return reply;
}
```

Kod instanciranja objekta u konstruktor se šalje agent, predložak poruke kako bi ponašanje čitalo samo poruke zahtjeva za pretplatom te implementaciju `SubscriptionManager` sučelja.

Predložak poruke kreiran je na sljedeći način:

```
MessageTemplate subscriptionTemplate= MessageTemplate.and(
        MessageTemplate.and(
                MessageTemplate.MatchPerformative(ACLMessage.SUBSCRIBE),

MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_SUBSCRIBE)
        ),
        MessageTemplate.and(
                MessageTemplate.MatchLanguage(codec.getName()),
                MessageTemplate.MatchOntology(ontology.getName())));
```

Ponašanje se agentu dodaje na sljedeći način:

```
addBehaviour(new PriceSubscriptionResponder(this, subscriptionTemplate,
this));
```

Metoda `register` iz sučelja `SubscriptionManager` uzima AID pošiljatelja te ga zajedno sa `Subscription` objektom dodaje u mapu `Map<AID, Subscription> participants`.

```
@Override
public boolean register(Subscription subscription) throws RefuseException,
NotUnderstoodException {

    AID newId = subscription.getMessage().getSender();
    participants.put(newId, subscription);

    return false;
}
```

Dodano je zatim i ponašanje koje će odjaviti pretplaćenog agenta ukoliko taj agent završi svoj životni ciklus. Za to je iskorištena klasa `AMSSubscriber` kod koje je nadjačana metoda `installHandlers` te je dodan sljedeći programski kod:

```
handlersTable.put(IntrospectionOntology.DEADAGENT, (EventHandler) ev -> {
    DeadAgent da = (DeadAgent) ev;
    AID id = da.getAgent();
    if (participants.containsKey(id)) {
        try {
            deregister(participants.get(id));
        } catch (FailureException e) {
            Logger.println(e.toString());
        }
    }
});
```

Zadnje dodano ponašanje je `PriceChangeCheckBehaviour` koje nasljeđuje `CyclicBehaviour`. Ovo ponašanje prima poruke od pametnog brojila kada se želi pokrenuti kućanski aparat. Sadržaj poruke je instanca klase `HouseHoldingDemand`. Temeljem zahtijevane snage i pomoću metode `calculatePrice`, elektrana računa kolika će biti cijena ukoliko se kućanski aparat uključi te šalje odgovor čiji sadržaj je instanca klase `PriceSignal`. Kako bi se osiguralo da čita samo poruke vezane uz provjeru cijene, kod ovog ponašanja nadjačana je metoda `onStart` te je u njoj kreiran predložak poruke koji je proslijeđen `receive` metodi.

Predložak poruke kreiran je unutar `onStart` metode na sljedeći način:

```
template = MessageTemplate.and(
        MessageTemplate.and(
                MessageTemplate.and(
                        MessageTemplate.MatchLanguage(codec.getName()),
                        MessageTemplate.MatchOntology(ontology.getName())
                ),
                MessageTemplate.and(

MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_QUERY),

MessageTemplate.MatchPerformative(ACLMessage.QUERY_IF)
                )
        ),
        MessageTemplate.MatchConversationId("CheckPriceChange")
);
```

Metoda `action` putem `receive` metode i predloška poruke zaprima poruku te uzima njezin sadržaj koji je zapravo objekt klase `Demand`. Iz `Demand` objekta uzima zahtijevanu snagu te poziva metodu `calculatePrice` kojoj prosljeđuje zbroj zahtijevane snage i trenutne snage koju elektrana poslužuje.

```
@Override
public void action() {

    final ACLMessage message = receive(template);
    if (message == null) {
        block();
        return;
    }

    final AID sender = message.getSender();

    try {
        ContentElement content =
getContentManager().extractContent(message);
        Demand demand = (Demand) ((Action) content).getAction();

        final BigDecimal price = calculatePrice(currentPowerConsumption +
demand.getPower());

        final ACLMessage reply = message.createReply();
        reply.setPerformative(ACLMessage.INFORM);
        reply.setLanguage(codec.getName());
        reply.setOntology(ontology.getName());
        final PriceSignal priceCheckSignal = new PriceSignal();
        priceCheckSignal.setMinPrice(priceSignal.getMinPrice());
        priceCheckSignal.setMaxPrice(priceSignal.getMaxPrice());
        priceCheckSignal.setCurrentPrice(price.floatValue());
        getContentManager().fillContent(reply, new Action(sender,
priceCheckSignal));

        send(reply);
```

```
    } catch (Codec.CodecException | OntologyException e) {
        Logger.println(e.toString());
    }

}
```

Dvije su metode za računanje cijena. Metoda `calculatePrice` računa cijenu za snagu proslijeđenu kao ulazni argument metode te metoda `calculateCurrentPrice` koja računa cijenu za trenutnu snagu i koja samo poziva `calculatePrice` kojoj kao ulazni parametar šalje trenutnu snagu koju elektrana poslužuje.

```
private void calculateCurrentPrice() {
    final BigDecimal newCurrentPrice =
calculatePrice(currentPowerConsumption);
    setCurrentPrice(newCurrentPrice);
}

private BigDecimal calculatePrice(Integer powerConsumption) {
    return
priceStep.multiply(BigDecimal.valueOf(powerConsumption)).add(BigDecimal.val
ueOf(priceSignal.getMinPrice()));
}
```

Kada se dogodi bilo kakva promjena u cijeni, elektrana obavještava sve agente koji su se pretplatili na nju. Za to koristi mapu `Map<AID, Subscription> participants` kroz koju prolazi te pozivajući metodu `sendPriceSignal` svakom pretplatniku šalje poruku kojoj je sadržaj objekt klase `PriceSignal`.

```
private void sendPriceSignal(Subscription subscription) {

    try {

        final ACLMessage subscriptionMessage = subscription.getMessage();
        final AID receiver = subscriptionMessage.getSender();
        ACLMessage reply = subscriptionMessage.createReply();
        reply.setPerformative(ACLMessage.INFORM);
        getContentManager().fillContent(reply, new Action(receiver,
priceSignal));
        subscription.notify(reply);

    } catch (Codec.CodecException | OntologyException e) {
        Logger.println(e.toString());
    }
```

7.4.2. Agent SmartMeter

Klasa `SmartMeter` predstavlja agent pametnog brojila. Ovaj agent se pretplaćuje na elektranu te prima informacije o cijeni. Također, prima poruke od uređaja kada se određeni uređaj želi uključiti te temeljem korisnikovih postavki i trenutne cijene električne energije odlučuje može li se uređaj uključiti, može li prebaciti opskrbu na obnovljivi izvor energije ili će staviti uređaj u red čekanja. Osim toga, pametno brojilo prati i trenutnu i ukupnu potrošnju.

Ova klasa nasljeđuje klasu `Agent` i implementira sučelje `SmartMeterManager` koje pruža metode za postavljanje visine cijene do koje želi da se uređaji uključuju, ispis povijesti potrošnje i dohvaćanje trenutne potrošnje.

```
void setPriceSelection(PriceSelectionEnum priceSelection);

void printConsumptionHistory();

Integer getCurrentPower();
```

Metoda `setup`, kao i kod agenta `PowerPlanttility` (7.4.1), prvo registrira ontologiju i jezik te sučelje, a zatim poziva `register` metodu. Obzirom da je u tom dijelu sve većinom isto kao i kod agenta `PowerPlanttility` (7.4.1), taj dio koda ovdje neće biti prikazan.

Ovom agentu dodano je pet ponašanja: `PriceSubscriptionInitiator`, `ProposeStartResponder`, `ApplianceStartedBehaviour`, `ApplianceEndedBehaviour` i `SolarCapacityCheckBehaviour`.

`PriceSubscriptionInitiator` nasljeđuje klasu `SubscriptionInitiator`. Ona šalje zahtjev za pretplatom elektrani te prima informacije o trenutnoj cijeni. Temeljem tih informacija i korisnikove postavke za visinu cijene, računa koja mu je maksimalna cijena do koje će dozvoljavati da se uređaji uključuju.

```
class PriceSubscriptionInitiator extends SubscriptionInitiator {

    boolean receivedResponse = false;

    public PriceSubscriptionInitiator(Agent a, ACLMessage msg) {
        super(a, msg);
    }

    @Override
    protected void handleInform(ACLMessage inform) {
        readPriceSignalMessage(inform);
        receivedResponse = true;
    }
```

```
    @Override
    protected void handleFailure(ACLMessage failure) {
        System.out.println("Something went wrong. Subscription to Power
Plant has failed");
        receivedResponse = true;
        getAgent().doDelete();
    }

    @Override
    protected void handleAllResponses(Vector responses) {
        if (!receivedResponse && responses.isEmpty()) {
            System.out.println("Timeout occurred for subscription!" +
getLocalName());
            getAgent().doDelete();
        }
    }
}
```

Dodavanje ponašanja i kreiranje poruke za pretplatu izgledaju ovako:

```
ACLMessage subscription = new ACLMessage(ACLMessage.SUBSCRIBE);
subscription.setProtocol(FIPANames.InteractionProtocol.FIPA_SUBSCRIBE);
subscription.setLanguage(codec.getName());
subscription.setOntology(ontology.getName());
subscription.addReceiver(powerPlantUtility);
Calendar calendar = Calendar.getInstance();
calendar.add(Calendar.SECOND, 15);
subscription.setReplyByDate(calendar.getTime());

addBehaviour(new PriceSubscriptionInitiator(this, subscription));
```

Metoda `readPriceSignalMessage` čita poruku te poziva metode za kalkulaciju maksimalne dozvoljene cijene za pokretanje uređaja i za kreiranje zapisa o potrošnji. Na kraju vrši provjeru cijene te poziva metode koje odlučuju može li se startati neki uređaj iz reda čekanja ili može li se prebaciti potrošnja na drugi izvor.

```
private void readPriceSignalMessage(ACLMessage msg) {

    try {

        ContentElement content = getContentManager().extractContent(msg);
        PriceSignal priceSignal = (PriceSignal) ((Action)
content).getAction();
        plantMaxPrice =
BigDecimal.valueOf(priceSignal.getMaxPrice().doubleValue()).setScale(2,
RoundingMode.HALF_UP);
        plantMinPrice =
BigDecimal.valueOf(priceSignal.getMinPrice().doubleValue()).setScale(2,
RoundingMode.HALF_UP);
        currentPrice =
BigDecimal.valueOf(priceSignal.getCurrentPrice().doubleValue()).setScale(2,
RoundingMode.HALF_UP);

        calculateMaxStartPrice();
```

```
        System.out.println("\nTrenutna cijena: " + currentPrice);
        System.out.println("\nMax. cijena za start: " + maxStartPrice);

        Date newRecordStartDate = new Date();
        createConsumptionRecord(newRecordStartDate);

        if (isPowerPlantPriceLowerOrEqualToMaxStartPrice()) {
            doPowerPlantCheckOnLowPriceSignal();
        } else if (powerSource.equals(PowerSourceEnum.POWER_PLANT)) {
            doSolrCheckOnHighPriceSignal();
        }

    } catch (Codec.CodecException | OntologyException e) {
        Logger.println(e.toString());
    }

}
```

Za visinu cijene korisnik može odabrati 5 razina: niska, srednje niska, srednja, srednje visoka i visoka. Temeljem tog odabira te informacija dobivenih od elektrane metoda `calculateMaxStartPrice` računa maksimalnu dozvoljenu cijenu do koje će dopuštati uređajima da se uključuju.

```
private void calculateMaxStartPrice() {

    final int numberOfLevels = PriceSelectionEnum.values().length;
    final int level = priceSelection.ordinal() + 1;
    if (level == numberOfLevels) {
        maxStartPrice = plantMaxPrice;
    } else {
        final BigDecimal priceStep =
plantMaxPrice.subtract(plantMinPrice).divide(BigDecimal.valueOf(numberOfLev
els), 2, RoundingMode.HALF_UP);
        maxStartPrice =
plantMinPrice.add(priceStep.multiply(BigDecimal.valueOf(level)));
    }
}
```

Ukoliko je cijena niža od maksimalne dozvoljene cijene, poziva se metoda `doPowerPlantCheckOnLowPriceSignal`. Ona provjerava je li trenutni korišteni izvor elektrana te ukoliko je provjera može li pokrenuti neki od uređaja iz reda čekanja. Ukoliko je trenutni izvor solarna energija, provjerava može li trenutnu potrošnju prebaciti na elektranu i može li onda pokrenuti neki od uređaja iz reda čekanja.

```
private void doPowerPlantCheckOnLowPriceSignal() throws
Codec.CodecException, OntologyException {

    if (powerSource.equals(PowerSourceEnum.POWER_PLANT)) {
        startQueuedAppliancesOnPowerPlantIfPossible();
    } else {
        Demand demand = new Demand();
        demand.setPower(currentPower);
```

```
        if (checkPowerPlantPrice(demand)) {
            changePowerSource(PowerSourceEnum.POWER_PLANT);
            startQueuedAppliancesOnPowerPlantIfPossible();
        }
    }
}
```

Metoda `startQueuedAppliancesOnPowerPlantIfPossible` ide kroz red čekanja uređaja te provjerava s elektranom hoće li se cijena povećati ukoliko se uređaj uključi. Ukoliko može uključiti uređaj šalje mu poruku putem metode `sendStartMessageToAppliance`.

```
private void startQueuedAppliancesOnPowerPlantIfPossible() throws
Codec.CodecException, OntologyException {
    final Iterator<Map.Entry<AID, HouseHoldingDemand>> entryIterator =
applianceQueue.entrySet().iterator();
    while (entryIterator.hasNext()) {
        final Map.Entry<AID, HouseHoldingDemand> entry =
entryIterator.next();
        final HouseHoldingDemand queueDemand = entry.getValue();
        if (checkPowerPlantPrice(queueDemand)) {
            System.out.println("\n" + entry.getKey().getLocalName() + " iz
reda čekanja može započeti zbog pada cijene. Izvor - " +
(powerSource.equals(PowerSourceEnum.SOLAR) ? "Solar" : "Elektrana"));
            sendStartMessageToAppliance(entry.getKey(), queueDemand);
            entryIterator.remove();
        }
    }
}
```

Metoda za provjeru hoće li se cijena promijeniti uključivanjem uređaja kreira poruku kojoj kao sadržaj šalje objekt klase `Demand` sa ukupnom snagom kojom će opteretiti elektranu ukoliko se uređaj uključi te čeka odgovor. Ukoliko odgovor ne stigne unutar 5 sekundi metoda vraća `false`, a ukoliko odgovor stigne, čita cijenu iz odgovora te uspoređuje sa maksimalnom dozvoljenom cijenom do koje se uređaji mogu pokretati.

```
private boolean checkPowerPlantPrice(Demand demand) throws
Codec.CodecException, OntologyException {
    ACLMessage req = new ACLMessage(ACLMessage.QUERY_IF);
    req.setProtocol(FIPANames.InteractionProtocol.FIPA_QUERY);
    req.setConversationId("CheckPriceChange");
    req.setLanguage(codec.getName());
    req.setOntology(ontology.getName());
    getContentManager().fillContent(req, new Action(powerPlantUtility,
demand));
    req.addReceiver(powerPlantUtility);

    send(req);

    MessageTemplate msgTemplate = MessageTemplate.and(
            MessageTemplate.and(
```

```
                    MessageTemplate.and(
                            MessageTemplate.MatchLanguage(codec.getName()),
MessageTemplate.MatchOntology(ontology.getName())
                    ),
                    MessageTemplate.and(
                            MessageTemplate.MatchSender(powerPlantUtility),
MessageTemplate.MatchPerformative(ACLMessage.INFORM)
                    )
            ),
            MessageTemplate.MatchConversationId("CheckPriceChange")
    );

    final ACLMessage aclMessage = blockingReceive(msgTemplate, 5000);

    if (aclMessage == null) {
        return false;
    }

    ContentElement content =
getContentManager().extractContent(aclMessage);
    PriceSignal priceSignal = (PriceSignal) ((Action) content).getAction();

    return
BigDecimal.valueOf(priceSignal.getCurrentPrice()).compareTo(maxStartPrice)
< 1;
}
```

Metoda koja služi za slanje informacije uređaju da se može uključiti kreira poruku kojoj kao sadržaj šalje objekt klase `Switch` u kojem je polje `action` stavljeno na `true`.

```
private void sendStartMessageToAppliance(AID receiver, HouseHoldingDemand
houseHoldingDemand) throws Codec.CodecException, OntologyException {
    Switch switchAction = new Switch();
    switchAction.setAction(true);
    ACLMessage reply = new ACLMessage(ACLMessage.INFORM);
    reply.setConversationId("SwitchAction");
    reply.setLanguage(codec.getName());
    reply.setOntology(ontology.getName());
    reply.addReceiver(receiver);
    getContentManager().fillContent(reply, new Action(receiver,
switchAction));

    send(reply);
}
```

Ukoliko je provjera u metodi `readPriceSignalMessage` pokazala da je cijena veća od maksimalne dozvoljene cijene za pokretanje uređaja, a trenutni izvor je elektrana, pokušati će prebaciti opskrbu na solarni izvor pomoću metode `doSolrCheckOnHighPriceSignal`.

```
private void doSolrCheckOnHighPriceSignal() throws Codec.CodecException,
OntologyException {
```

```
        if (checkSolarCapacity(null)) {
            System.out.println("\nPrebacivanje na Solar zbog povećanja
cijene");
            changePowerSource(PowerSourceEnum.SOLAR);
            startQueuedAppliancesOnSolarIfPossible();
        }
    }
```

Metoda `startQueuedAppliancesOnSolarIfPossible` je gotovo ista kao i `startQueuedAppliancesOnPowerPlantIfPossible` uz razliku da se u njoj provjerava može li solar opskrbiti trenutnu potražnu pomoću metode `checkSolarCapacity`.

Metoda `checkSolarCapacity` traži od DF agenta AID solarnog panela te mu šalje poruku za provjeru u kojoj šalje listu svih zahtjeva uređaja koje treba opskrbiti.

```
private boolean checkSolarCapacity(HouseHoldingDemand newDemand) throws
Codec.CodecException, OntologyException {
        ServiceDescription sd = new ServiceDescription();
        sd.setType("SolarPanel");

        DFAgentDescription template = new DFAgentDescription();
        template.addServices(sd);

        SearchConstraints all = new SearchConstraints();
        all.setMaxResults(1L);

        DFAgentDescription[] results;
        Set<AID> agents = new LinkedHashSet<>();

        try {
            results = DFService.search(SmartMeter.this, template, all);
            for (DFAgentDescription result : results) {
                agents.add(result.getName());
            }
        } catch (FIPAException fe) {
            fe.printStackTrace();
        }

        final jade.util.leap.List list = new jade.util.leap.ArrayList(new
ArrayList<>(working.values()));
```

```
        if (newDemand != null) {
            list.add(newDemand);
        }
        final SolarDemand solarDemand = new SolarDemand();
        solarDemand.setDemandList(list);

        ACLMessage req = new ACLMessage(ACLMessage.QUERY_IF);
        req.setProtocol(FIPANames.InteractionProtocol.FIPA_QUERY);
        req.setConversationId("CheckSolarCapacity");
        req.setLanguage(codec.getName());
        req.setOntology(ontology.getName());
        final AID receiver = agents.iterator().next();
        getContentManager().fillContent(req, new Action(receiver,
solarDemand));
        req.addReceiver(receiver);

        send(req);

        MessageTemplate msgTemplate = MessageTemplate.and(
                MessageTemplate.and(
                        MessageTemplate.and(

MessageTemplate.MatchLanguage(codec.getName()),

MessageTemplate.MatchOntology(ontology.getName())
                        ),
                        MessageTemplate.and(
                                MessageTemplate.MatchSender(receiver),

MessageTemplate.MatchPerformative(ACLMessage.INFORM)
                        )
                ),
                MessageTemplate.MatchConversationId("CheckSolarCapacity")
        );

        final ACLMessage aclMessage = blockingReceive(msgTemplate, 5000);
        return Boolean.valueOf(aclMessage.getContent());
    }
```

Metoda `changePowerSource` ne radi ništa pametno već samo mijenja vrijednost polja `powerSource` te dodaje redak u listu povijesti o potrošnji.

Drugo dodano ponašanje je `ProposeStartResponder` koje nasljeđuje klasu `ProposeResponder`, a dodano je na sljedeći način:

```
MessageTemplate proposeStartTemplate = MessageTemplate.and(
        MessageTemplate.and(
                MessageTemplate.and(
                        MessageTemplate.MatchLanguage(codec.getName()),
                        MessageTemplate.MatchOntology(ontology.getName())
                ),
                MessageTemplate.and(

MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_PROPOSE),

MessageTemplate.MatchPerformative(ACLMessage.PROPOSE)
                )
        ),
        MessageTemplate.MatchConversationId("ProposeStart")
);
addBehaviour(new ProposeStartResponder(this, proposeStartTemplate));
```

Metoda u kojoj se odvija sva logika unutar ovog ponašanja je `prepareResponse`. Ona je nadjačana i implementirana tako da vrši provjeru je li trenutni izvor elektrana te ako je provjerava je li trenutna cijena niža od maksimalne dozvoljene. Ukoliko je niža provjerava s elektranom hoće li se promijeniti cijena i preći maksimalnu dozvoljenu ukoliko se uređaj upali. Ukoliko će se cijena povećati, vrši provjeru može li trenutnu potrošnju prebaciti na solarni izvor.

Ukoliko je cijena već viša od dozvoljene provjerava može li prebaciti trenutnu potrošnju na solarni izvor.

Ukoliko je opskrba električne energije već prebačena na solarni izvor, provjerava ima li dovoljno energije da se posluži i ovaj uređaj.

Ako na bilo koji način može dozvoliti uključivanje uređaja, šalje mu poruku u kojoj ga obavještava da prihvaća prijedlog i da se uređaj može upaliti. U suprotnom mu šalje odbijenicu te dodaje uređaj na listu čekanja pomoću metode `scheduleAppliance`.

```
@Override
protected ACLMessage prepareResponse(ACLMessage propose) throws
NotUnderstoodException, RefuseException {

    final AID sender = propose.getSender();
    ACLMessage reply = propose.createReply();

    try {
        ContentElement content =
getContentManager().extractContent(propose);
        HouseHoldingDemand houseHoldingDemand = (HouseHoldingDemand)
((Action) content).getAction();
```

```
        boolean start = false;

        if (powerSource.equals(PowerSourceEnum.POWER_PLANT)) {
            if (isPowerPlantPriceLowerOrEqualToMaxStartPrice()) {
                if (checkPowerPlantPrice(houseHoldingDemand)) {
                    System.out.println("\n" + sender.getLocalName() + "
može započeti odmah. Izvor - Elektrana.");
                    start = true;
                } else {
                    if (checkSolarCapacity(houseHoldingDemand)) {
                        System.out.println("\nPrebacivanje na izvor Solar.
" + sender.getLocalName() + " može započeti odmah. Izvor - Solar.");
                        changePowerSource(PowerSourceEnum.SOLAR);
                        start = true;
                    } else {
                        System.out.println("\n" + sender.getLocalName() + "
dodan u red čekanja zbog mogućeg povećanja cijene.");
                    }
                }
            } else {
                if (working.isEmpty()) {
                    if (checkSolarCapacity(houseHoldingDemand)) {
                        changePowerSource(PowerSourceEnum.SOLAR);
                        start = true;
                        System.out.println("\nPrebacivanje na izvor Solar.
" + sender.getLocalName() + " može započeti odmah. Izvor - Solar.");
                    } else {
                        System.out.println("\n" + sender.getLocalName() + "
dodan u red čekanja zbog previsoke trenutne cijene.");
                    }
                } else {
                    System.out.println("\n" + sender.getLocalName() + "
dodan u red čekanja zbog previsoke trenutne cijene.");
                }
            }
        } else {
            if (checkSolarCapacity(houseHoldingDemand)) {
                System.out.println("\n" + sender.getLocalName() + " može
započeti odmah. Izvor - Solar.");
                start = true;
            } else {
                System.out.println("\n" + sender.getLocalName() + " dodan u
red čekanja zbog premale proizvodnje panela.");
            }
        }

        if (start) {
            reply.setPerformative(ACLMessage.ACCEPT_PROPOSAL);
        } else {
            reply.setPerformative(ACLMessage.REJECT_PROPOSAL);
            scheduleAppliance(sender, houseHoldingDemand);
        }

    } catch (Codec.CodecException | OntologyException e) {
        Logger.println(e.toString());
        reply.setPerformative(ACLMessage.REJECT_PROPOSAL);
    }
```

```
    return reply;
}
```

Metoda `scheduleAppliance`, kao što je ranije navedeno, dodaje zahtjev u red čekanja. Red čekanja predstavljen je mapom `Map<AID, HouseHoldingDemand> applianceQueue` koja povezuje AID agenta i njegov zahtjev za uključivanjem.

```
private void scheduleAppliance(AID sender, HouseHoldingDemand
houseHoldingDemand) {
    applianceQueue.put(sender, houseHoldingDemand);
}
```

Treće dodano ponašanje je `ApplianceStartedBehaviour` koje nasljeđuje `CyclicBehaviour`. Ono služi za primanje informacije od uređaja kada se uređaj stvarno pokrenuo. Ovo ponašanje nadjačava `onStart` i `action` metode. U metodi `onStart` kreira se predložak za poruke koje će primati, a u `action` metodi se prima poruku i poziva metoda `handleApplianceStart`.

```
@Override
public void onStart() {
    template = MessageTemplate.and(
            MessageTemplate.and(
                    MessageTemplate.MatchLanguage(codec.getName()),
                    MessageTemplate.MatchOntology(ontology.getName())
            ),
            MessageTemplate.and(

MessageTemplate.MatchConversationId("ApplianceStarted"),
                    MessageTemplate.MatchPerformative(ACLMessage.INFORM)
            )
    );
}


@Override
public void action() {

    final ACLMessage message = receive(template);
    if (message == null) {
        block();
        return;
    }

    try {
```

```
        ContentElement content =
getContentManager().extractContent(message);
        HouseHoldingDemand houseHoldingDemand = (HouseHoldingDemand)
((Action) content).getAction();

        handleApplianceStart(message.getSender(), houseHoldingDemand);

    } catch (Codec.CodecException | OntologyException e) {
        Logger.println(e.toString());
    }
}
```

Metoda `handleApplianceStart` jednostavno dodaje uređaj u mapu pokrenutih uređaja `Map<AID, HouseHoldingDemand> working`, računa trenutnu potrošnju te dodaje redak u listu povijesti potrošnje.

Četvrto dodano ponašanje je `ApplianceEndedBehaviour` koje nasljeđuje `CyclicBehaviour`. Ovo ponašanje služi za primanje obavijesti od uređaja kada je uređaj završio s radom. Ovo ponašanje nadjačava `onStart` i `action` metode. U `onStart` metodi kreira se predložak za poruke koje će primati, a u `action` metodi se prima poruku i poziva metoda `endAppliance`.

```
@Override
public void onStart() {
    template = MessageTemplate.and(
            MessageTemplate.MatchConversationId("ApplianceEnded"),
            MessageTemplate.MatchPerformative(ACLMessage.INFORM)
    );
}


@Override
public void action() {

    final ACLMessage message = receive(template);
    if (message == null) {
        block();
        return;
    }

    final AID sender = message.getSender();
    endAppliance(sender);
}
```

Metoda `endAppliance` kalkulira novu trenutnu potrošnju, miče agenta iz mape koja sadrži trenutno pokrenute agente (polje `working`) te kreira zapis u povijesti potrošnje.

Peto dodano ponašanje je `SolarCapacityCheckBehaviour` koje nasljeđuje `TickerBehaviour`. Ovo ponašanje se okida periodički u razmaku koje mu je zadano kroz konstruktor.

U ovom ponašanju nadjačana je metoda `onTick` te se u njoj, svakih pola sata (u ovom radu skalirano na 30 sekundi) provjerava može li se prebaciti potrošnja na solarni izvor ukoliko se struja trenutno uzima iz elektrane i cijena je prevelika ili, ukoliko se struja trenutno uzima iz solarnog izvora, može li se startati koji uređaj iz reda čekanja.

```
@Override
protected void onTick() {

    if (working.isEmpty() && applianceQueue.isEmpty()) {
        return;
    }

    if (powerSource.equals(PowerSourceEnum.POWER_PLANT) &&
(isPowerPlantPriceHigherThenMaxStartPrice() || !applianceQueue.isEmpty()))
{
        try {
            if (checkSolarCapacity(null)) {
                changePowerSource(PowerSourceEnum.SOLAR);
                startQueuedAppliancesOnSolarIfPossible();
            }
        } catch (final Codec.CodecException | OntologyException e) {
            Logger.println(e.getLocalizedMessage());
        }

        return;
    }

    if (powerSource.equals(PowerSourceEnum.SOLAR)) {
        try {
            startQueuedAppliancesOnSolarIfPossible();
        } catch (final Codec.CodecException | OntologyException e) {
            Logger.println(e.getLocalizedMessage());
        }
    }
}
```

7.4.3. Agent HouseHoldingAppliance

Klasa `HouseHoldingAppliance` je klasa apstraktna klasa koja nasljeđuje klasu `Agent`, sadrži cijelu logiku potrebnu za funkcioniranje agenta kućanskog aparata no sadrži i dvije apstraktne metode, `populateProgramsMap` i `createServiceDescription` koje se pozivaju unutar `register` metode.

Ovu klasu nasljeđuju klase koje predstavljaju sušilicu (`Dishwasher`), perilicu rublja (`WashingMachine`) i sušilicu za rublje (`Dryer`).

Metoda `populateProgramsMap` služi za postavljanje mogućih programa rada, naziva i duljine trajanja. Dok `createServiceDescription` kreira `ServiceDescription` za tog agenta.

```
@Override
void populateProgramsMap() {
    programs.put("Program_1", 60);
    programs.put("Program_2", 90);
    programs.put("Program_3", 120);
}

@Override
void createServiceDescription() {
    ServiceDescription sd = new ServiceDescription();
    sd.setType("Dryer");
    sd.setName(getLocalName());
    dfd.addServices(sd);
}
```

Metode `setup` i `register` su gotovo iste kao i kod drugih agenata pa njihov kod neće biti prikazan. Kao ulazni parametar ovom agentu prosljeđuje se snaga u Watima.

Unutar `setup` metode dodano je samo jedno ponašanje, `StartMessageHandleBehaviour` koje nasljeđuje `CyclicBehaviour`. Ovo ponašanje prima poruku od pametnog brojila kada se može uključiti ukoliko nije bilo uključeno odmah već je bilo dodano u red čekanja. U `onStart` metodi kreira se predložak poruka koje će čitati, a u `action` metodi dohvaća se poruka, popunjava se vrijeme pokretanja u `HouseHoldingDemand` objektu, uređaj se starta te se obavještava brojilo da je uređaj pokrenut.

```
@Override
public void onStart() {
    template = MessageTemplate.and(
            MessageTemplate.and(
                    MessageTemplate.MatchLanguage(codec.getName()),
                    MessageTemplate.MatchOntology(ontology.getName())
            ),
            MessageTemplate.and(
                    MessageTemplate.MatchConversationId("SwitchAction"),
                    MessageTemplate.MatchPerformative(ACLMessage.INFORM)
            )
    );
}

@Override
public void action() {

    final ACLMessage message = receive(template);
    if (message == null) {
        block();
        return;
```

```
    }

    try {
        ContentElement content =
getContentManager().extractContent(message);
        Switch switchAction = (Switch) ((Action) content).getAction();
        final boolean action = switchAction.getAction();
        if (action) {
            final Integer programDuration = programs.get(pickedProgram);

            HouseHoldingDemand demand = new HouseHoldingDemand();
            demand.setStartDate(new Date());
            demand.setDuration(programDuration);
            demand.setPower(power);

            startWorking(demand, message.getSender());
        }
    } catch (Codec.CodecException | OntologyException e) {
        Logger.println(e.toString());
    }
}
```

Metoda `startWorking` šalje poruku brojilu da je startana pomoću metode `sendApplianceStartedMessage` te kreira *timer* koji će se pokrenuti nakon što istekne vrijeme rada uređaja definirano za odabrani program. Kada to vrijeme istekne, poslati će obavijest pametnom brojilu pomoću metode `sendApplianceEndedMessage` da je završio.

```
private void startWorking(HouseHoldingDemand demand, AID receiver) {

    try {
        sendApplianceStartedMessage(receiver, demand);

        final Timer timer = new Timer();
        timer.schedule(new TimerTask() {
            @Override
            public void run() {

                sendApplianceEndedMessage(receiver);
                pickedProgram = null;

            }
        }, demand.getDuration() * 1000L);

    } catch (Codec.CodecException | OntologyException e) {
        Logger.println(e.toString());
    }
}
```

Metoda za obavještavanje brojila o početku rada uređaja priprema poruku u kojoj ponovno, kao sadržaj, šalje isti objekt klase `HouseHoldingDemand` koji je bio poslan kao sadržaj zahtjeva za paljenjem, uz razliku da prvo popuni `startDate` polje.

```
private void sendApplianceStartedMessage(AID receiver, HouseHoldingDemand
demand) throws Codec.CodecException, OntologyException {

    ACLMessage startInfoMsg = new ACLMessage(ACLMessage.INFORM);
    startInfoMsg.setLanguage(codec.getName());
    startInfoMsg.setOntology(ontology.getName());
    startInfoMsg.setConversationId("ApplianceStarted");
    startInfoMsg.addReceiver(receiver);
    Date startDate = new Date();
    demand.setStartDate(startDate);
    getContentManager().fillContent(startInfoMsg, new Action(receiver,
demand));
    send(startInfoMsg);
}
```

Metoda za obavještavanje brojila o kraju rada šalje poruku bez sadržaja, no brojilo će znati na što se poruka odnosi prema vrijednosti polja `conversationId`.

```
private void sendApplianceEndedMessage(AID receiver) {
    ACLMessage endInfoMsg = new ACLMessage(ACLMessage.INFORM);
    endInfoMsg.setConversationId("ApplianceEnded");
    endInfoMsg.addReceiver(receiver);
    send(endInfoMsg);
}
```

`HouseHoldingAppliance` implementira sučelje `HouseHoldingApplianceManager` koje pruža metodu za pokretanje uređaja `demandStart`. Toj metodi prosljeđuje se ime programa koji se želi pokrenuti te ona traži od DFa AID pametnog brojila kojemu šalje zahtjev za uključivanjem tako da kreira poruku i ponašanje `ProposeStartInitiator` koje dodaje agentu.. Obzirom da je pretraživanje DFa prikazano kod opisivanja `SmartMeter` agenta (7.4.2) kod metode `checkSolarCapacity`, ovdje će taj dio metode biti preskočen te će biti prikazano samo kreiranje poruke i ponašanja.

```
HouseHoldingDemand houseHoldingDemand = new HouseHoldingDemand();
houseHoldingDemand.setDuration(programs.get(program));
houseHoldingDemand.setPower(power);

ACLMessage proposeMsg = new ACLMessage(ACLMessage.PROPOSE);
proposeMsg.setLanguage(codec.getName());
proposeMsg.setOntology(ontology.getName());
proposeMsg.setConversationId("ProposeStart");
Calendar calendar = Calendar.getInstance();
calendar.add(Calendar.SECOND, 15);
proposeMsg.setReplyByDate(calendar.getTime());
```

```
final AID receiver = agents.iterator().next();
getContentManager().fillContent(proposeMsg, new Action(receiver,
houseHoldingDemand));
proposeMsg.addReceiver(receiver);


addBehaviour(new ProposeStartInitiator(this, proposeMsg,
houseHoldingDemand));
```

Ponašanje `ProposeStartInitiator` nasljeđuje `ProposeInitiator`. Ono šalje poruku sa zahtjevom za uključenje pametnom brojilu te ukoliko dobije pozitivan odgovor pokreće uređaj pomoću već opisane metode `startWorking`.

```
class ProposeStartInitiator extends ProposeInitiator {

    private boolean receivedResponse = false;
    private HouseHoldingDemand demand;

    private ProposeStartInitiator(Agent a, ACLMessage msg,
HouseHoldingDemand demand) {
        super(a, msg);
        this.demand = demand;
    }

    @Override
    protected void handleAcceptProposal(ACLMessage acceptProposal) {

        receivedResponse = true;
        startWorking(demand, acceptProposal.getSender());
    }

    @Override
    protected void handleAllResponses(Vector responses) {

        if (!receivedResponse && responses.isEmpty()) {
            System.out.println("Timeout occurred!" + getLocalName());
            pickedProgram = null;
            return;
        }

        final ACLMessage message = (ACLMessage) responses.get(0);
        if (ACLMessage.FAILURE == message.getPerformative()) {
            System.out.println("Something went wrong with starting " +
getLocalName());
            pickedProgram = null;
        }
    }
}
```

7.4.4. Agent SolarPanel

Klasa `SolarPanel` nasljeđuje klasu `Agent` te predstavlja agenta koji upravlja solarnim izvorom energije. Ideja je da ovaj agent skladišti električnu energiju ukoliko trenutno nije taj koji opskrbljuje kuću te zaprima upite od pametnog brojila i računa može li zadovoljiti potražnju. Ova logika nije implementirana već ova klasa implementira sučelje `SolarPanelManager` koje pruža metodu `setHasEnoughCapacity` kroz koju je moguće hoće li na upit odgovoriti pozitivno ili negativno.

Ono što je implementirano kod ovog agenta je dohvaćanje lokacije te dohvaćanje predviđanja proizvodnje električne energije u sljedeća 24 sata. Ispis predviđanja vrši se pomoću metode `printProductionPrediction` iz sučelja `SolarPanelManager`.

Metode `setup` i `register` su ponovno veoma slične već ranije opisanima pa neće biti prikazane. Vrijedi napomenuti ovaj agent kod startanja prima argumente s adresom na kojoj se nalazi, azimutom i vršnom proizvodnom snagom panela te koristi te podatke prvo kako bi dohvatio lokaciju preko Googleovog geocoding REST servisa, a zatim i kako bi svaka 24 sata dohvaćao predviđanje o proizvodnji u sljedećih 48 sati.

Dohvaćanje podataka o lokaciji događa se unutar metode `fetchLatitudeAndLongitude`.

```
private void fetchLatitudeAndLongitude(String address) {
    Client client = ClientBuilder.newClient();
    WebTarget webResource = client.target("https://maps.google.com/")
            .path("maps/api/geocode/json");
    try {
        webResource = webResource.queryParam("address",
                URLEncoder.encode(address, "UTF-8"));
    } catch (UnsupportedEncodingException e) {
        Logger.println(e.getMessage());
        takeDown();
        return;
    }
    webResource = webResource.queryParam("key", "AIzaSyCmxvfY-
RT2TdY8556L6Att-Asik-uQpEs");

    String response =
webResource.request(MediaType.APPLICATION_JSON).get(String.class);
    JsonReader reader = null;
    try {

        reader = Json.createReader(new StringReader(response));
        JsonObject jo = reader.readObject();

        if (jo.getJsonArray("results").isEmpty()) {
            takeDown();
        }
```

```
        JsonObject obj = jo.getJsonArray("results")
                .getJsonObject(0)
                .getJsonObject("geometry")
                .getJsonObject("location");

        latitude = obj.getJsonNumber("lat").toString();
        latitude = obj.getJsonNumber("lng").toString();
    } finally {
        if (reader != null) {
            reader.close();
        }
    }
}
```

Ponašanje koje dohvaća predviđanje o proizvodnji energije je `TickerBehavior` i unutar njega se poziva metoda `fetchPowerProductionPrediction`.

```
addBehaviour(new TickerBehaviour(this, 48 * 60 * 60 * 1000) {
    @Override
    protected void onTick() {
        fetchPowerProductionPrediction();
    }
});


private void fetchPowerProductionPrediction() {
    Client client = ClientBuilder.newClient();
    WebTarget webResource = client.target("https://api.forecast.solar/")
            .path("estimate/")
            .path(latitude + "/")
            .path(longitude + "/")
            .path(declination + "/")
            .path(azimuth + "/")
            .path(kwp);

    String response =
webResource.request(MediaType.APPLICATION_JSON).get(String.class);

    JsonReader reader = null;
    try {
        reader = Json.createReader(new StringReader(response));
        JsonObject jo = reader.readObject();
        final String results = jo.get("result").toString();

        Gson gson = new GsonBuilder().setDateFormat("yyyy-MM-dd
HH:mm:ss").create();
        prediction = gson.fromJson(results, SolarPrediction.class);

    } finally {
        if (reader != null) {
            reader.close();
        }
    }
}
```

Prikaz rezultata napravljen je u poglavlju 7.5.

Drugo dodano ponašanje je `CapacityCheckBehaviour` koje nasljeđuje `CyclicBehaviour` i koje prima zahtjeve za provjerom od pametnog brojila. Ponašanje u `onStart` metodi kreira predložak za primanje poruka, a u `action` metodi čita poruku te poziva metodu `isSufficientForCurrentSupply` za provjeru i odgovara brojilu. Već je napomenuto da `isSufficientForCurrentSupply` nije implementirano na neki pametan način već samo za potrebe simulacije vraća ono što je postavljeno putem metode `setHasEnoughCapacity(boolean hasEnoughCapacity)`.

```
@Override
public void onStart() {
    super.onStart();
    template = MessageTemplate.and(
            MessageTemplate.and(
                    MessageTemplate.and(
                            MessageTemplate.MatchLanguage(codec.getName()),

MessageTemplate.MatchOntology(ontology.getName())
                    ),
                    MessageTemplate.and(

MessageTemplate.MatchProtocol(FIPANames.InteractionProtocol.FIPA_QUERY),

MessageTemplate.MatchPerformative(ACLMessage.QUERY_IF)
                    )
            ),
            MessageTemplate.MatchConversationId("CheckSolarCapacity")
    );
}

@Override
public void action() {

    final ACLMessage message = receive(template);
    if (message == null) {
        block();
        return;
    }

    try {
        ContentElement content =
getContentManager().extractContent(message);
        SolarDemand demand = (SolarDemand) ((Action) content).getAction();

        final boolean sufficientForCurrentSupply =
isSufficientForCurrentSupply(demand);

        final ACLMessage reply = message.createReply();
        reply.setPerformative(ACLMessage.INFORM);
        reply.setLanguage(codec.getName());
        reply.setOntology(ontology.getName());
        reply.setContent("" + sufficientForCurrentSupply);

        send(reply);
```

```
    } catch (Codec.CodecException | OntologyException e) {
        Logger.println(e.toString());
    }
}
```

7.5. Primjer izvođenja

U ovom poglavlju bit će prikazane dvije simulacije rada sustava pametne zgrade. Raspored zahtjeva za uključenjem uređaja i vremenski periodi u kojima dolaze bit će potpuno isti kao i njihova snaga te odabrani programi rada. Prvo će pristići zahtjev perilice suđa koja ima snagu od 1500 W, a odabrani program rada traje 90 minuta. Nakon toga pristići će zahtjev sušilice koja ima snagu 2000 W, a odabrani program rada traje 120 minuta. Zadnji zahtjev pristići će od perilice rublja koja ima snagu 450 W, a odabrani program rada traje 60 minuta. Postavke elektrane, također će biti iste. Minimalna cijena struje, postavljena na agentu elektrane, bit će 2 kune, a maksimalna 5 kuna po kilowatsatu. Radi lakše simulacije maksimalna snaga elektrane bit će 5000 W, a vremenska trajanja programa bit će skalirana na sekunde (60 minuta = 60 sekundi). Razlike u simulacijama bit će u odabranoj postavci za visinu cijene na pametnom brojilu te u tome hoće li u nekom trenutku biti moguće prebaciti opskrbu na solarni izvor.

7.5.1. Simulacija 1

U ovoj simulaciji postavka za visinu cijene, na agentu pametnog brojila, postavljena je na vrijednost MEDIUM, a solarni panel nikada neće imati dovoljnu proizvodnju električne energije kako bi zadovoljio potražnju.

Odmah nakon uključenja pametno brojilo prema informacijama dobivenim od elektrane izračunava maksimalnu dopuštenu cijenu za pokretanje uređaja od 3.80 kn. Početna cijena struje je 2.00 kn, a uskoro dolazi do povećanja cijene na 2.57 kn. Nakon toga perilica suđa prva zahtijeva početak rada. Obzirom da se cijena neće povećati iznad 3.80 kn ukoliko se upali ovaj uređaj, pametno brojilo dopušta rad. Nakon početka rada, cijena struje se opet povećava te sada iznosi 3.47 kn. Slijedi zahtjev za radom od strane sušilice. Pokretanje sušilice povećalo bi cijenu preko 3.80 kn stoga pametno brojilo odbija početak rada te stavlja sušilicu u red čekanja. Nakon što perilica suđa završi s radom, cijena struje pada na 2.57 kn te sušilica može započeti s radom. Zatim cijena struje opet raste, sada na 3.77 kn. Dok sušilica još uvijek radi, pristiže zahtjev za uključenjem perilice rublja.

Uključivanjem perilice rublja cijena bi porasla preko maksimalne dozvoljene cijene od 3.80 kn pa pametno brojilo dodaje perilicu rublja u red čekanja. Nakon nekog vremena cijena pada na 3.32 kn te sada perilica rublja može započeti s radom. Cijena sada raste na 3.59 kn. Nakon toga sušilica završava s radom te cijena pada na 2.39 kn. Na kraju još završava i perilica rublja te cijena dodatno pada na 2.12 kn. Kompletan opisani ispis događaja prikazan je na slici 21.

```
19:02:26

Trenutna cijena: 2.00

Max. cijena za start: 3.80

19:02:27

Trenutna cijena: 2.57

Max. cijena za start: 3.80

19:02:27 - Agent Perilica suđa može započeti odmah. Izvor - Elektrana.

19:02:27 - Agent Perilica suđa započeo s radom. Izvor - Elektrana

19:02:27

Trenutna cijena: 3.47

Max. cijena za start: 3.80

19:02:47 - Agent Sušilica dodan u red čekanja zbog mogućeg povećanja cijene.

19:03:57 - Agent Perilica suđa završio s radom. Izvor - Elektrana

19:03:57

Trenutna cijena: 2.57

Max. cijena za start: 3.80

19:03:57 - Agent Sušilica iz reda čekanja može započeti zbog pada cijene. Izvor - Elektrana

19:03:57 - Agent Sušilica započeo s radom. Izvor - Elektrana

19:03:57

Trenutna cijena: 3.77

Max. cijena za start: 3.80

19:04:47 - Agent Perilica rublja dodan u red čekanja zbog mogućeg povećanja cijene.

19:05:17

Trenutna cijena: 3.32

Max. cijena za start: 3.80

19:05:17 - Agent Perilica rublja iz reda čekanja može započeti zbog pada cijene. Izvor - Elektrana

19:05:17 - Agent Perilica rublja započeo s radom. Izvor - Elektrana

19:05:17

Trenutna cijena: 3.59

Max. cijena za start: 3.80

19:05:57 - Agent Sušilica završio s radom. Izvor - Elektrana

19:05:57

Trenutna cijena: 2.39

Max. cijena za start: 3.80

19:06:17 - Agent Perilica rublja završio s radom. Izvor - Elektrana

19:06:17

Trenutna cijena: 2.12

Max. cijena za start: 3.80
```

Slika 21. Ispis događaja tijekom rada sustava

Osim ispisa događaja moguće je vidjeti i kolika je bila cijena struje i potrošnja u Watima u pojedinom razdoblju što prikazuje slika 22.

```
19:02:26 - 19:02:27 --> 0W, cijena po kwh: 2.00 kn
19:02:27 - 19:02:27 --> 0W, cijena po kwh: 2.57 kn
19:02:27 - 19:02:27 --> 1500W, cijena po kwh: 2.57 kn
19:02:27 - 19:03:57 --> 1500W, cijena po kwh: 3.47 kn
19:03:57 - 19:03:57 --> 0W, cijena po kwh: 3.47 kn
19:03:57 - 19:03:57 --> 0W, cijena po kwh: 2.57 kn
19:03:57 - 19:03:57 --> 2000W, cijena po kwh: 2.57 kn
19:03:57 - 19:05:17 --> 2000W, cijena po kwh: 3.77 kn
19:05:17 - 19:05:17 --> 2000W, cijena po kwh: 3.32 kn
19:05:17 - 19:05:17 --> 2450W, cijena po kwh: 3.32 kn
19:05:17 - 19:05:57 --> 2450W, cijena po kwh: 3.59 kn
19:05:57 - 19:05:57 --> 450W, cijena po kwh: 3.59 kn
19:05:57 - 19:06:17 --> 450W, cijena po kwh: 2.39 kn
19:06:17 - 19:06:17 --> 0W, cijena po kwh: 2.39 kn
19:06:17 - 19:06:22 --> 0W, cijena po kwh: 2.12 kn
```

Slika 22. Prikaz potrošnje u Watima i cijene struje kroz različita vremenska razdoblja

Osim prikaza potrošnje u Watima i cijene struje po kilowat satu, moguće je vidjeti i kako je rastao račun za struju tako da korisnik može kontrolirati potrošnju i unaprijed znati koliki račun će dobiti. Ispis je u ovom radu napravljen u csv obliku kako bi bilo lakše generirati grafikon za analizu rada. Ovaj ispis prikazan je na slici 23.

```
Vrijeme,Potrosnja
19:02:27,0.00
19:03:57,7.81
19:05:17,17.86
19:05:57,23.72
19:06:17,24.08
19:06:22,24.08
```

Slika 23. Prikaz rasta računa za struju

7.5.2. Simulacija 2

U ovoj simulaciji postavka za visinu cijene, na agentu pametnog brojila, postavljena je na vrijednost MEDIUM_LOW, jednu razinu niže nego u simulaciji 1, a solarni panel će u jednom trenutku biti u mogućnosti zadovoljiti potražnju za električnom energijom.

Prema dobivenim podacima od elektrane pametno brojilo izračunava maksimalnu dozvoljenu cijenu za uključivanje uređaja od 3.20 kn. Početna cijena je 2.00 kn, no nakon nekog vremena raste na 2.57 kn. Kao i prvoj simulaciji, prvo dolazi zahtjev za uključivanjem od strane perilice suđa, a zatim od strane sušilice. Oba uređaja dodana su u red čekanja jer bi njihovim uključenjem cijena struje porasla preko 3.20 kn. Nakon njih dolazi zahtjev od strane perilice rublja čije uključenje ne bi uzrokovalo povećanje cijene preko 3.20 kn te pametno brojilo odobrava uključivanje uređaja. Cijena struje raste na 2.84 kn, a nakon nekog vremena pada na 2.39 kn. Nakon što perilica suđa završi s radom cijena pada na 2.12 kn pa brojilo provjerom utvrđuje da je moguće uključiti perilicu suđa. Perilica suđa započinje s radom te cijena raste na 3.02 kn. Nakon što perilica suđa završi cijena pada na 2.12 kn. Pametno brojilo provjerom utvrđuje da pomoću obnovljivog izvora energije može opskrbiti sušilicu te joj šalje poruku za uključenjem i prebacuje izvor električne energije na solarnu energiju. Sušilica obavlja svoj rad pogonjena električnom energijom dobivenom od solarnih panela. Slika 24. prikazuje opisani ispis događaja tijekom rada sustava.

```
19:10:52

Trenutna cijena: 2.00

Max. cijena za start: 3.20

19:10:53

Trenutna cijena: 2.57

Max. cijena za start: 3.20

19:10:53 - Agent Perilica suđa dodan u red čekanja zbog mogućeg povećanja cijene.

19:11:13 - Agent Sušilica dodan u red čekanja zbog mogućeg povećanja cijene.

19:13:13 - Agent Perilica rublja može započeti odmah. Izvor - Elektrana.

19:13:13 - Agent Perilica rublja započeo s radom. Izvor - Elektrana

19:13:13

Trenutna cijena: 2.84

Max. cijena za start: 3.20

19:13:43

Trenutna cijena: 2.39

Max. cijena za start: 3.20

19:14:13 - Agent Perilica rublja završio s radom. Izvor - Elektrana

19:14:13

Trenutna cijena: 2.12

Max. cijena za start: 3.20

19:14:13 - Agent Perilica suđa iz reda čekanja može započeti zbog pada cijene. Izvor - Elektrana

19:14:13 - Agent Perilica suđa započeo s radom. Izvor - Elektrana

19:14:13

Trenutna cijena: 3.02

Max. cijena za start: 3.20

19:15:43 - Agent Perilica suđa završio s radom. Izvor - Elektrana

19:15:43

Trenutna cijena: 2.12

Max. cijena za start: 3.20

19:15:52 - Agent Sušilica započeo s radom. Izvor - Solar

19:17:52 - Agent Sušilica završio s radom. Izvor - Solar
```

Slika 24. Ispis događaja tijekom rada sustava

Potrošnja i kretanje cijena u pojedinom razdoblju prikazana je na slici 25., a kretanje računa za struju prikazano je na slici 26.

```
19:10:52 - 19:10:53 --> 0W, cijena po kwh: 2.00 kn
19:10:53 - 19:13:13 --> 0W, cijena po kwh: 2.57 kn
19:13:13 - 19:13:13 --> 450W, cijena po kwh: 2.57 kn
19:13:13 - 19:13:43 --> 450W, cijena po kwh: 2.84 kn
19:13:43 - 19:14:13 --> 450W, cijena po kwh: 2.39 kn
19:14:13 - 19:14:13 --> 0W, cijena po kwh: 2.39 kn
19:14:13 - 19:14:13 --> 0W, cijena po kwh: 2.12 kn
19:14:13 - 19:14:13 --> 1500W, cijena po kwh: 2.12 kn
19:14:13 - 19:15:43 --> 1500W, cijena po kwh: 3.02 kn
19:15:43 - 19:15:43 --> 0W, cijena po kwh: 3.02 kn
19:15:43 - 19:15:52 --> 0W, cijena po kwh: 2.12 kn
19:15:52 - 19:15:52 --> 0W, cijena po kwh: 2.12 kn
19:15:52 - 19:17:52 --> 2000W, cijena po kwh: 0 kn (Solar)
19:17:52 - 19:17:57 --> 0W, cijena po kwh: 0 kn (Solar)
```

Slika 25. Prikaz potrošnje u Watima i cijene struje kroz različita vremenska razdoblja

```
Vrijeme,Potrosnja
19:10:53,0.00
19:13:13,0.00
19:13:43,0.64
19:14:13,1.18
19:15:43,7.98
19:15:52,7.98
19:17:52,7.98
19:17:57,7.98
```

Slika 26. Prikaz rasta računa za struju

Kako je u ovoj simulaciji korišten solarni izvor energije, ovdje će biti prikazano kako izgleda predviđanje proizvodnje koje se dohvaća putem REST web servisa.

```
{
  "watts": {
    "Sun Sep 08 06:17:00 CEST 2019": 0,
    "Sun Sep 08 06:31:00 CEST 2019": 12,
    "Sun Sep 08 06:45:00 CEST 2019": 102,
    . . .
    "Mon Sep 09 06:18:00 CEST 2019": 0,
    "Mon Sep 09 06:32:00 CEST 2019": 18,
    "Mon Sep 09 06:45:00 CEST 2019": 114,
```

```
    . . .
    "Mon Sep 09 19:25:00 CEST 2019": 0
  },
  "watt_hours": {
    "Sun Sep 08 06:17:00 CEST 2019": 0,
    "Sun Sep 08 06:31:00 CEST 2019": 0,
    "Sun Sep 08 06:45:00 CEST 2019": 24,
    "Sun Sep 08 07:00:00 CEST 2019": 66,
    "Sun Sep 08 08:00:00 CEST 2019": 618,
    . . .
    "Sun Sep 08 19:27:00 CEST 2019": 17760,
    "Mon Sep 09 06:18:00 CEST 2019": 0,
    "Mon Sep 09 06:32:00 CEST 2019": 6,
    "Mon Sep 09 06:45:00 CEST 2019": 30,
    "Mon Sep 09 07:00:00 CEST 2019": 90,
    "Mon Sep 09 08:00:00 CEST 2019": 1086,
    . . .
    "Mon Sep 09 19:25:00 CEST 2019": 36060
  },
  "watt_hours_day": {
    "Sun Sep 08 00:00:00 CEST 2019": 17760,
    "Mon Sep 09 00:00:00 CEST 2019": 36060
  }
}
```


7.6. Analiza rezultata

Analiza rezultata bit će obavljena s ekonomskog stajališta kako bi se prikazalo koliko korisnik može uštediti. Za svaku simulaciju bit će napravljena usporedba između pametne i obične zgrade. Bit će napravljena analiza kretanja cijene struje, potrošnje u Watima i kretanja računa za struju

Kako bi izgledala potrošnja i kretanje cijena te kretanje računa za struju kada ova zgrada ne bi bila pametna zgrada te kada bi svi uređaji počeli raditi i trošiti struju istog trena kada ih korisnik uključi prikazano je na slici 27. i slici 28.

```
19:21:26 - 19:21:27 --> 0W, cijena po kwh: 2.00 kn
19:21:27 - 19:21:27 --> 0W, cijena po kwh: 2.57 kn
19:21:27 - 19:21:27 --> 1500W, cijena po kwh: 2.57 kn
19:21:27 - 19:21:47 --> 1500W, cijena po kwh: 3.47 kn
19:21:47 - 19:21:47 --> 3500W, cijena po kwh: 3.47 kn
19:21:47 - 19:22:57 --> 3500W, cijena po kwh: 4.67 kn
19:22:57 - 19:22:57 --> 2000W, cijena po kwh: 4.67 kn
19:22:57 - 19:23:47 --> 2000W, cijena po kwh: 3.77 kn
19:23:47 - 19:23:47 --> 0W, cijena po kwh: 3.77 kn
19:23:47 - 19:23:47 --> 450W, cijena po kwh: 3.77 kn
19:23:47 - 19:23:47 --> 450W, cijena po kwh: 2.57 kn
19:23:47 - 19:24:17 --> 450W, cijena po kwh: 2.84 kn
19:24:17 - 19:24:47 --> 450W, cijena po kwh: 2.39 kn
19:24:47 - 19:24:47 --> 0W, cijena po kwh: 2.39 kn
19:24:47 - 19:24:52 --> 0W, cijena po kwh: 2.12 kn
```

Slika 27. Prikaz potrošnje u Watima i cijene kroz različita razdoblja kod obične zgrade

```
Vrijeme,Potrosnja
19:21:27,0.00
19:21:47,1.73
19:22:57,20.80
19:23:47,27.08
19:24:17,27.72
19:24:47,28.26
19:24:52,28.26
```

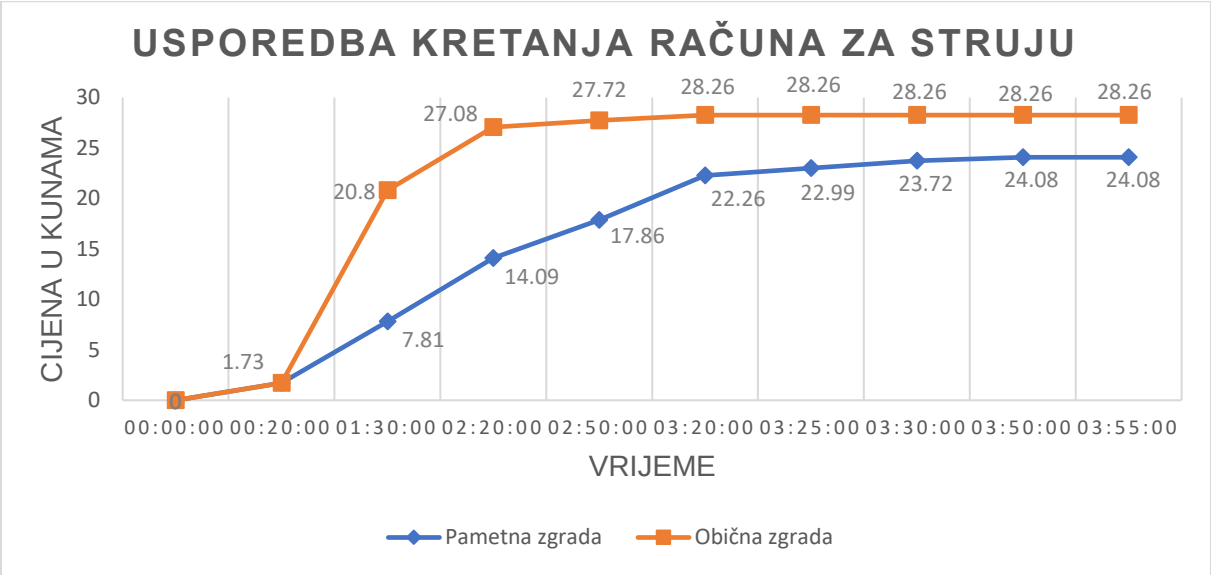
Slika 28. Prikaz rasta računa za struju kod obične zgrade

7.6.1. Simulacija 1

Zbog mogućih povećanja cijene, sušilica i perilica rublja bili su dodani u red čekanja te su tek naknado dobili zeleno svijetlo od brojila i započeli s radom. Sušilica je čekala 1 sat i 20 minuta na početak rada, dok je perilica rublja čekala 20 minuta. Kod obične zgrade to nije bio slučaj te su svi uređaji počinjali s radom čim ih je korisnik uključio. Rezultat toga je da su uređaji u običnoj zgradi završili s radom nakon 3 sata i 20 minuta dok je kod pametne zgrade to trajalo 1 sat i 30 minuta dulje, odnosno sveukupno 3 sata i 50 minute.

Ukoliko se usporedi financijski aspekt, vidljivo je da nakon završetka rada svih uređaja u običnoj zgradi račun iznosi 28.26 kuna dok je kod pametne zgrade to 4.18 kn manje, odnosno ukupno 24.08 kn. Razlog je, naravno taj, što su kod obične zgrade svi uređaji krenuli s radom čim ih je korisnik uključio što je i dodatno utjecalo na rast cijene struje u tim trenucima.

Usporedba kretanja računa za vrijeme rada uređaja u pametnoj i u običnoj zgradi prikazana je sljedećim grafikonom:

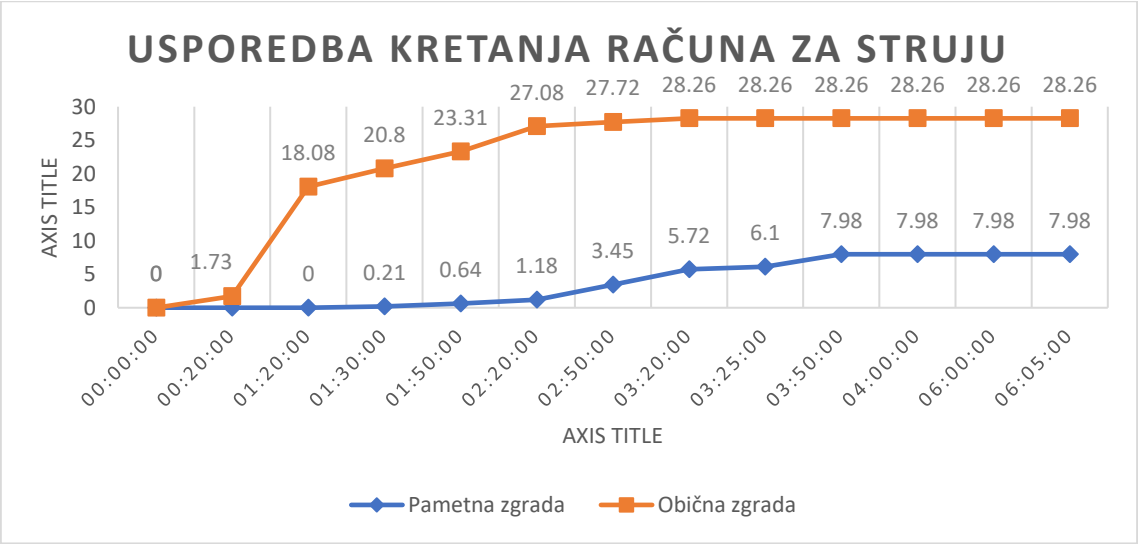


7.6.2. Simulacija 2

Zbog niže postavke za visinu cijene do koje se uređaji mogu paliti, kod ove simulacije brojilo nije dozvolilo početak rada perilici suđa i sušilici već ih je dodalo u red čekanja. Perilica rublja je mogla započeti s radom čim je poslala zahtjev, perilica suđa je morala čekati na smanjenje cijene 3 sata i 20 minuta, a sušilica se uključila nakon 4 sata i 40 minuta kada je bilo dovoljno električne energije iz solarnog izvora kako bi se sušilica napajala za vrijeme rada. Zbog svega navedenog, bilo je potrebno 6 sati da bi svi uređaji u pametnoj zgradi odradili svoj posao što je 2 sata i 40 minuta dulje nego kod obične zgrade te 2 sata i 10 minuta dulje nego kod pametne zgrade u prvoj simulaciji.

Kako je postavka za visinu cijene do koje brojilo može dopustiti rad u ovoj simulaciji bila niža nego u prethodnoj simulaciji, a jedan uređaj je za svoj rad bio napajan pomoću solarnog izvora, račun za struju je dosta niži nego kod obične zgrade i ušteda je veće nego u prvoj simulaciji. Račun na kraju simulacije iznosi 7.98 kn što je 20.28 kn manje od obične zgrade i 16.10 kn manje od pametne zgrade u prvoj simulaciji.

Usporedba kretanja računa za vrijeme rada uređaja u pametnoj i u običnoj zgradi prikazana je sljedećim grafikonom:



7.7. Usporedba s drugim sustavima

Mnogo je radova koji se bave tematikom pametnih zgrada koje sudjeluju u DR programima te svaki od njih pruža različite implementacije i različiti pristup problemu. Ovdje ćemo usporediti neke od tih implementacija sa implementacijom u ovom radu.

Neke od implementacija baziraju se na dobivanju cijena za buduća vremenska razdoblja i planiranju potrošnje pa tako Althaher, Mancarella i Mutale u svom radu [20] implementiraju sustav u kojem se cijene dobivaju u naprijed, za sljedeća 24 sata i temeljem toga sustav planira kada se koji uređaj može uključiti. Također, sustav prima ažurirane informacije o cijeni svaka 3 sata te tada provodi reorganizaciju u planiranju. Corelo, Moajes i Baringo [21] implementiraju sustav koji dobiva cijenu za svaki sat, 10 minuta u naprijed, te 5 minuta prije početka sata šalje informacije o planiranoj potrošnji. Qian, Zhang, Huang i Wu [22] opisuju sličan sustav, no kod njih pružatelj usluge i korisnik razmjenjuju poruke o cijeni i potražnji u budućim vremenskim periodima. Broj poruka je ograničen, a cilj je da se postigne dogovor o potražnji i cijeni na obostrano zadovoljstvo.

U ovom radu ne postoji planiranje paljenja uređaja niti se cijene primaju u naprijed već isključivo u stvarnom vremenu te se samo odgađa uključivanje uređaja dok cijena ponovno ne padne. Time je smanjena kompleksnost sustava i algoritma potrebnog za planiranje rada uređaja, ali i kompleksnost komunikacije. Ovaj je zato jednostavniji za

implementaciju te može izbjeći problem konstantnog reorganiziranja planiranog paljenja uređaja te omogućava korisniku da sam određuje do koje cijene će pametno brojilo dopuštati paljenje uređaja. S druge strane, u sustavu implementiranom u ovom radu nije moguće unaprijed znati kada će se koji uređaj uključiti dok je kod gore navedenih sustava to moguće.

Sustav koji predstavljaju LeMay, Nelli i Gunter [23] ima mnoge sličnosti sa sustavom implementiranim u ovom radu. U njihovom radu sustav, također reagira na cijene u stvarnom vremenu, uz razliku da pametni uređaji mogu sami odlučivati o uključivanju temeljem cijene dok za druge, ne pametne, uređaje odlučuje pametno brojilo. Ovakav pristup može biti dobar jer omogućava da se za pametne uređaje postave različite postavke cijene te da oni sami odlučuju o uključivanju i time rasterete pametno brojilo. S druge strane, može biti kompliciranо za korisnika jer treba podesiti postavke svakog pametnog uređaja posebno te ne može jednostavno odrediti granicu u cijeni iznad koje ne želi paliti uređaje. Ovaj sustav nema provjeru hoće li se cijena povećati ukoliko se neki uređaj uključi te prilikom pada cijene može uključiti uređaje i time podignuti cijenu iznad željene granice.

Ni jedan od navedenih sustava ne uzima u obzir obnovljive izvore energije koji su danas sve popularniji te se financijski potiče njihova implementacija. Korištenjem obnovljivih izvora, moguće je skladištiti energiju te ju koristiti kasnije ili ju koristiti odmah kako se generira te time dodatno smanjiti opterećenje električne mreže, račun za struju te vrijeme čekanja do uključivanja nekog uređaja.

Svi navedeni primjeri imaju jedno centralno mjesto koje vrši komunikaciju s elektranom i koje se u nekim od gore navedenih radova, kao i u ovom radu, naziva pametno brojilo dok ga drugi nazivaju kontrolna jedinica. Sličnost s ovim radom je i u tome što je to centralno mjesto, u neku ruku, pretplaćeno na elektranu koja u određenim intervalima ili pri svakoj promjeni cijene, ovisno u implementaciji, šalje informacije o cijeni.

U ovom radu cilj je bio postići sustav koji je jednostavniji za implementaciju, manje kompleksan, a samim time jeftiniji te jednostavniji za korištenje i shvaćanje od strane korisnika, stoga, kako je ranije navedeno, nema planiranja paljenja uređaja ili kompleksne komunikacije s elektranom već samo dodavanje uređaja u red čekanja i provjera kako će uključivanje uređaja utjecati na trenutnu cijenu. Ova implementacija je i manjeg opsega te obuhvaća samo uređaje koji se ne mogu prekidati u radu dok u drugim radovima autori pokrivaju i uređaje koji se smiju prekinuti ili čija se potrošnja može prilagoditi te time dobivaju fleksibilniji sustav.

8. Zaključak

U ovom radu obrađene su teme agenta, višeagentnih sustava te pametne zgrade i pametne (električne) mreže. Objašnjene su karakteristike i ciljevi pametne zgrade te uloga agenata i agentnog programiranja u implementiranju takvog sustava. Fokus je, u radu, stavljen na energetsku učinkovitost, to jest na smanjenje potrošnje električne energije te posljedično i smanjenje računa za struju.

Korištenjem agenata, agentnog programiranja i programskog okvira JADE koji pruža sve potrebne funkcionalnosti za razvoj i pokretanje agenata napravljena je implementacija pametne zgrade kao višeagentnog sustava u kojem se, pomoću komunikacije između elektrane i pametnog brojila te komunikacije između pametnog brojila i uređaja unutar zgrade, odlučuje hoće li se pojedini uređaj upaliti ili dodati u red čekanja. Implementacija je napravljena na način da korisnik može definirati postavku za maksimalnu dozvoljenu cijenu temeljem koje će pametno brojilo donositi odluke. Pametna zgrada ima i obnovljivi izvor energije na koji je moguće prebaciti opskrbu ukoliko on može zadovoljiti trenutnu potrebu. Ovakav pristup implementaciji pametne (električne) zgrade rješava ranije navedeni problem vršne potražnje te nepotrebnog generiranja električne energije. Na taj način, ovaj sustav, smanjuje zagađenje, ali i troškove izgradnje i održavanja elektrane i električne mreže. Posljedično, smanjuje se i cijena električne energije koju će pružatelj naplaćivati, a osim toga, reagiranjem pametnog brojila na promjene u cijeni korisnik se dodatno smanjuju troškovi. Kada se uzme u obzir i korištenje obnovljivih izvora energije koji mogu skladištiti energiju te povremeno opskrbljivati korisnika, ili slati (prodavati) proizvedenu energiju u električnu mrežu, ovakav sustav postaje još isplativiji.

Prikazom implementacije i rada sustava te kasnijom analizom rezultata moguće je zaključiti da se pomoću agentnog programiranja i povezivanja agenata u višeagentni sustav može, poprilično jednostavno, implementirati sustav koji će biti energetski učinkovit te koji će pametnim odlukama rasporediti potrošnju na obostrano zadovoljstvo, kako korisnika, tako i elektrane. Analizom rezultata prikazano je kako takav sustav može pozitivno utjecati na ekonomske aspekte korisnika, a iz dobivenih rezultata da se zaključiti i kako bi ovakav sustav utjecao na samu električnu mrežu te na elektrane. Može se zaključiti kako bi se primjenom ovakve tehnologije, osim računa za struju, smanjila i vršna potražnja te samim time cijena izgradnje i održavanja elektrana i potrebne infrastrukture, a što je najvažnije smanjilo bi se i zagađenje okoliša.

Popis literature

[1] S. Russell i P. Norvig, *Artificial Intelligence A Modern Approach Third Edition*, Prentice Hall, 2009.

[2] M. Wooldridge, „Intelligent Agents" u *A Modern Approach to Distributed Modern Approach to Artificial Intelligence*, Massachusetts Institute of Technology, 1999, str. 27-78.

[3] P. Anthony, K .O. Chin, K. S. Gan, R. Alfred i D. Lukose, „Agent Architecture: An Overview" 01.2014. [Na internetu]. Dostupno: https://www.researchgate.net/publication/275643980_Agent_Architecture_An_Overview. [Pristupano 25.06.2019].

[4] A. Dorri, R. Jurdak i S. S. Kanhere, „Multi-Agent Systems: A survey" 04.2018. [Na internetu]. Dostupno: https://www.researchgate.net/publication/324847369_Multi-Agent_Systems_A_survey. [Pristupano 25.06.2019].

[5] M. Schatten, „Inteligentni agenti" [Moodle]. Dostupno: https://elfarchive1718.foi.hr/mod/resource/view.php?id=23688. [Pristupano 25.06.2019].

[6] B. Morvaj, L. Lugaric i S. Krajcar, „Demonstrating Smart Buildings and Smart Grid features in a Smart Energy City" Siječanj 2011. [Na internetu]. Dostupno: https://www.researchgate.net/publication/252046548_Demonstrating_smart_buildings_and_smart_grid_features_in_a_smart_energy_city. [Pristupano 25 Veljača 2019].

[7] „Smart grid" u *Wikipedia, the Free Encyclopedia*. Dostupno: https://en.wikipedia.org/wiki/Smart_grid. [Pristupano 18.08.2019].

[8] Y. Bamberger i sur., „Vision and Strategy for Europe's Electricity Netowrks of the Future", Luksemburg: Office for Official Publications of the European Communities, 2006, str. 4.

[9] „Smart Grid" u *Techopedia*. Dostupno: https://www.techopedia.com/definition/692/smart-grid. [Pristupano 18.08.2019].

[10] „Smart Grid" u *SmartGrid.Gov*, U.S. Department of Energy: Office of Electricity Delivery and Energy Reliability. Dostupno: https://www.smartgrid.gov/the_smart_grid/smart_grid.html. [Pristupano 18.08.2019].

[11] S. Mohagheghi, J. Stoupis, Z. Wang, Z. Li i H. Kazemzadeh, „Demand Response Architecture: Integration into the Distribution Management System" 11.2010. [Na internetu]. Dostupno: https://www.researchgate.net/publication/224189806_Demand_Response_Architecture_Integration_into_the_Distribution_Management_System. [Pristupano 19.08.2019].

[12] „Smart Buildings" u *Gemalto a Thales company*, 07.03.2019. Dostupno: https://www.gemalto.com/m2m/markets/smart-buildings. [Pristupano 19.08.2019].

[13] „Energy Efficiency in Buildings“ u *Ghent University*. Dostupno: https://www.ugent.be/energhentic/en/topics/energy-efficiency/energyefficiencybuildings.htm. [Pristupano 24.08.2019].

[14] Capgemini, „Demand Response: A Decisive Breakthrough for Europe”, 2008. [Na internetu]. Dostupno: https://www.capgemini.com/wp-content/uploads/2017/07/Demand_Response__a_decisive_breakthrough_for_Europe.pdf. [Pristupano 24.08.2019].

[15] „JAVA Agent DEvelopment Framework” u *Tilab*, [Na internetu]. Dostupno: https://jade.tilab.com/. [Pristupano 25.08.2019].

[16] „JADE Programming Tutorial For Beginners“ 30.06.2009. [Na internetu]. Dostupno: https://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf. [Pristupano 25.08.2019].

[17] „Java Agent Development Framework” u *Wikipedia, the Free Encyclopedia* [Na internetu]. Dostupno: https://en.wikipedia.org/wiki/Java_Agent_Development_Framework. [Pristupano 25.08.2019].

[18] „JADE Programmer's Guide“ 08.04.2010. [Na internetu]. Dostupno: https://jade.tilab.com/doc/programmersguide.pdf. [Pristupano 25.08.2019].

[19] J. Vaucher i A. Ncho, „7. Using ontologies“ u *JADE Tutorial and Primer*, 09.2003., [Na internetu]. Dostupno: https://www.iro.umontreal.ca/~vaucher/Agents/Jade/Ontologies.htm. [Pristupano 25.08.2019].

[20] S. Althaher, P. Mancarella i J. Mutale, „Automated Demand Response from Home Energy Management System under Dynamic Pricing and Power and Comfort Constraints“, 2015, [Na internetu]. Dostupno: https://www.research.manchester.ac.uk/portal/files/20645950/POST-PEER-REVIEW-NON-PUBLISHERS.PDF. [Pristupano 15.09.2019].

[21] A. J. Conejo, J. M. Morales i L. Baringo, „Real-Time Demand Response Model“, 12.2010., [Na internetu]. Dostupno: https://www.adme.com.uy/db-docs/Docs_secciones/nid_150/DR_p007_RealTimeDemandResponseModel.pdf. [Pristupano 15.09.2019].

[22] L. P. Qian, Y. J. Zhang, J. Huang i Y. Wu, „Demand Response Management via Real-TimeElectricity Price Control in Smart Grids“, 07.2013., [Na internetu]. Dostupno: http://iranarze.ir/wp-content/uploads/2017/08/15-English-IranArze.pdf. [Pristupano 15.09.2019].

[23] M. LeMay, R. Nelli i C. A. Gunter, „An Integrated Architecture forDemand Response Communications and Control“, 01.2008., [Na internetu]. Dostupno: https://tcipg.org/sites/default/files/papers/LeMayNGG08.pdf. [Pristupano 15.09.2019].

Popis slika

Slika 1. Agent i okolina (Wooldrige [1]) 3
Slika 2. Percepcija i okolina (Wooldrige [2]) 6
Slika 3. Agent sa stanjima (Wooldrige [2]) 7
Slika 4. Shematski dijagram generičke BDI arhitekture (Wooldrige[2]) 11
Slika 5. Vertikalna i horizontalna arhitektura (Wooldrige [2]) 14
Slika 6. TouringMachines: Arhitektura s horizontalnim uslojavanjem (Wooldrige [2]) 15
Slika 7. InteRRap: Vertikalna arhitektura s dva prolaza (Wooldrige [2]) 16
Slika 8. Jednostavni refleksivni agent (Russell i Norvig [1]) 18
Slika 9. Refleksivni agent temeljen na modelu (Russell i Norvig [1]) 19
Slika 10. Agent na temelju modela, na temelju ciljeva (Russell i Norvig [1]) 20
Slika 11. Agent temeljen na modelu, temeljen na korisnosti (Russell i Norvig [1]) 21
Slika 12. Općeniti agent sa sposobnošću učenja (Russell i Norvig [1]) 22
Slika 13. Središnji agent, voditelj (engl. *Facilitator*; Dorri, Jurdak i Kanhere [4]) 24
Slika 14. Središnji agent, medijator (Dorri, Jurdak i Kanhere [4]) 24
Slika 15. Karakteristike tradicionalnog sustava (lijevo) i pametne mreže (desno) (Wikipedia [7]) 30
Slika 16. Pametna energetska kuća (Morvaj, Krajcar i Lugaric [6]) 34
Slika 17. JADE koncept [16] 38
Slika 18. Životni ciklus agenta prema definiciji FIPAe [18] 40
Slika 19. Dijagram klasa 45
Slika 20. Komunikacija u sustavu 46
Slika 21. Ispis događaja tijekom rada sustava 77
Slika 22. Prikaz potrošnje u Watima i cijene struje kroz različita vremenska razdoblja 78
Slika 23. Prikaz rasta računa za struju 78
Slika 24. Ispis događaja tijekom rada sustava 80
Slika 25. Prikaz potrošnje u Watima i cijene struje kroz različita vremenska razdoblja 81
Slika 26. Prikaz rasta računa za struju 81
Slika 27. Prikaz potrošnje u Watima i cijene kroz različita razdoblja kod obične zgrade 83
Slika 28. Prikaz rasta računa za struju kod obične zgrade 83