UDC: 004.43:004.42.045 Info M: str. 43-49

IMPLEMENTACIJA GENERIČKE DSP KOMPONENTE U SKLOPU TYPHOON HIL SOFTVERA ZA MODELOVANJE ŠEMA ENERGETSKIH SKLOPOVA
IMPLEMENTATION OF GENERIC DSP COMPONENT WITHIN TYPHOON HIL SOFTWARE FOR MODELING POWER CIRCUITS SCHEMES

Alen Suljkanović, Gordana Milosavljević, Dušan Majstorović, Igor Dejanović

REZIME: Potrošnja električne energije je u konstantnom porastu. Da bi se ovaj trend nesmetano nastavio neophodno je raditi na efikasnosti i pouzdanosti proizvodnje i distribucije električne energije. Jedna od neophodnih metoda koja se primenjuje da bi se postigao ovaj cilj jeste rigorozno i kontinualno testiranje svih komponenti elektroenergetskih sistema. Testiranje se može obavljati na realnom sistemu ali takav način testiranja je skup, spor i često dovodi do ugrožavanja materijalnih dobara i života ljudi. Zbog toga je u sve široj primeni vid testiranja koji se izvodi na posebnim uređajima, gde se realni sistem simulira modelom. Ovakav pristup je dosta efikasniji, jeftiniji i sigurniji od testiranja sistema "na živo". Kompanija TyphoonHIL proizvodi seriju HIL (Hardware-in-the-loop) uređaja za testiranje kontrolera u oblasti energetske elektronike i tehnologije digitalne obrade signala (Digital Signal Processing). Zbog širokog spektra primene tehnologije digitalne obrade signala nije moguće odrediti konačan skup DSP komponenti potrebnih krajnjem korisniku, zbog čega se javila potreba za implementacijom generičke DSP komponente. U ovom radu je predstavljena implementacija generičke DSP komponente u vidu CodeBlock komponente. Korisnik bira izgled komponente, te opisuje njenu funkcionalnost pomoću modifikovanog podskupa C jezika. CodeBlock komponenta je implementirana korišćenjem Arpeggio parsera uz oslonac na Jinja2 obrađivač šablona za generisanje koda i programski jezik Python. Rešenje predstavljeno u radu je integrisano u Typhoon Schematic editor, deo hardversko/softverskog skupa alata kompanije TyphoonHIL, što je omogućilo veću fleksibilnost alata i mogućnost proširenja od strane krajnjih korisnika.
KLJUČNE REČI: CodeBlock, generisanje koda, DSP, parsiranje, Python, PEG, gramatika, grafički editor, generisanje šeme
ABSTRACT: The need for electricity in the world is increasing, which necessitates more efficient production and distribution of electricity, which is achieved through rigorous and continuous testing of power systems of this type. Tests are performed on specific devices, where the real system is replaced with a model. This type of testing is much more efficient, cheaper and safer than testing the real system. The company TyphoonHIL produces series of HIL (Hardware-in-the-loop) devices for testing the controllers in the field of Power Electronics and technology of Digital Signal Processing. Because of the wide range of applications of Digital Signal Processing, it is not possible to determine a finite set of DSP components required by a user, which is why there was a need for the implementation of generic DSP component. This paper presents the implementation of a generic DSP component as CodeBlock component. The user selects the appearance of component and describes its functionality by using a modified subset of the C language. CodeBlock component is implemented by using Arpeggio parser, relying on Jinja2 template engine for code generation. The solution presented in this paper is integrated within Typhoon Schematic Editor, which is part of the TyphoonHIL hardware/software tool-chain. The solution makes Typhoon Schematic Editor a lot more flexible to the needs of the user.
KEY WORDS:CodeBlock, code generation, DSP, parsing , Python, PEG, grammar, graphic editor, scheme generation

1. UVOD

Porast potrošnje električne energije je konstantnom usponu, što uzrokuje i potrebu za povećanjem njene proizvodnje. Pri tome, sama proizvodnja električne energije i njena distribucija treba da bude što efikasnija i pouzdanija što nameće potrebu za rigoroznim i kontinualnim testiranjem sistema ovog tipa.

Zahvaljujući brzom razvoju energetske elektronike u poslednjih trideset godina, efikasnost energetskih sistema je znatno uvećana, posebno zbog razvoja poluprovodnika i mikroprocesora. U zadnje vremene, ova efikasnost je dodatno poboljšana razvojem invertora. U praksi se sve više koriste složene pretvaračke topologije i algoritmi upravljanja postaju sve složeniji. Sve ovo nameće potrebu za sve sofisticiranijim metodama za testiranje.

Testiranja ovakvih sistema „na živo" mogu da rezultuju ogromnom materijalnom štetom, pa čak i brojnim rizicima za operatere, ukoliko sistem nije dovoljno dobro dizajniran. Stoga, testiranje se uglavnom vrši na posebnim uređajima, gde se realni sistem zamenjuje modelom koji predstavlja dovoljno verodostojnu sliku realnog sistema. Testiranja ovog tipa su enormno efikasnija, jeftinija i sigurnija nego testiranje „na živo".

Seriju uređaja za testiranje energetskih postrojenja simulacijom u realnom vremenu (sa kašnjenjem od 1μs) proizvodi kompanija *TyphoonHIL*, lider u domenu energetske elektronike. Ovi uređaji se nazivaju HIL *(Hardware-in-the-loop)* uređajima. Oni predstavljaju hardversko/softversko rešenje čija je namena dizajniranje, razvoj i automatizovano testiranje kontrolera komercijalnih uređaja iz domena energetske elektronike. Pored energetske elektronike, *TyphoonHIL* sve više zalazi u domen digitalne obrade signala *(engl. Digital Signal Processing - DSP)* – oblasti koja je u poslednjih nekoliko dekada u ekspanziji.

U poslednjih trideset godina, upotreba digitalnih kontrolera u industriji je dramatično povećana. Razlog za to je pojava procesora digitalnih signala *(engl. Digital Signal Processors)*. Upotreba procesora digitalnih signala za realizaciju kontrolera prevazilazi neka od ograničenja koja su imali kontroleri bazirani na mikroprocesorima, jer arhitektura procesora digitalnih signala obezbeđuje da se aritmetičke operacije izvršavaju velikom brzinom i sa velikom tačnošću. Danas se procesori digitalnih signala koriste u velikom broju oblasti, počevši od telekomunikacija, preko medicine, digitalne obrade slike, istraživanja svemira, radara i sonara, i tako dalje [1].

Postoji više softverskih rešenja koja se koriste za projektovanje upravljačke logike i automatsko generisanje koda u oblasti energetske elektronike. Tri najpoznatija komercijalna rešenja su *Simulink, PSIM i PLECS,* pri čemu je *Simulink* najkvalitetnije i najčešće korišćeno rešenje. Iako navedeni komercijalni alati nude dosta široku paletu DSP komponenti, zbog širokog dijapazona primene tehnologije obrade digitalnih signala, nije moguće odrediti konačan skup komponenti koje bi određenom korisniku mogle biti potrebne. Stoga, ovi alati nude mogućnost da u zavisnosti od svojih potreba korisnici kreiraju svoje komponente, čime se premošćuje jaz između korisničkih potreba i funkcionalnosti koje nudi određeni alat. Jedan tip takvih komponenti je i *CodeBlock* komponenta predstavljena u ovom radu, koja je integrisana u *Schematic editor* kompanije *TyphoonHIL.*

2. DIZAJN CODEBLOCK KOMPONENTE

Arhitektura *TyphoonHIL* softverskih alata je bazirana na MVC *(Model-View-Controller)* obrascu. Model dele svi softverski alati, dok su grafički korisnički interfejs i kontroler implementirani za svaki alat zasebno. Model je sačinjen od klasa koje opisuju modele šeme, komponenata, žica i tako dalje. Skup klasa unutar modela i njihove međusobne interakcije predstavljaju radni okvir na koji se nadogradila implementacija *CodeBlock* komponente.

CodeBlock komponenta nasleđuje sve osobine standardne komponente, međutim ova komponenta se razlikuje od ostalih komponenti po tome što njena funkcionalnost nije predefinisana, nego njenu funkcionalnost opisuje korisnik. Korisnik je u mogućnosti da menja ovu funkcionalnost po želji tokom rada u *Typhoon Schematic editor*-u, čime je omogućena veća adaptivnost alata.

S obzirom na ovakav dizajn, pomoću ove komponente može se kreirati proizvoljna DSP komponenta, što *CodeBlock* komponentu čini svojevrsnim obrascem za kreiranje komponenti.

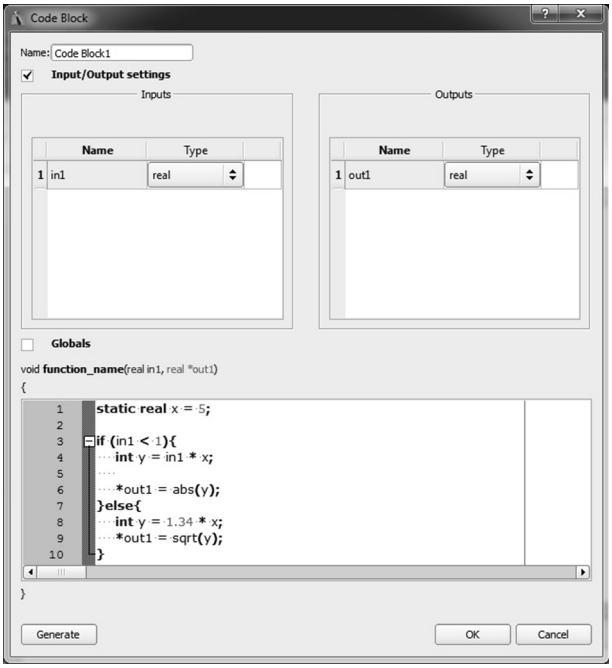
2.1. *Slučajevi korišćenja CodeBlock komponente*

U softverskom inženjerstvu, slučaj korišćenja je niz koraka, koji opisuju interakciju „aktera“ sa ostatkom sistema, koji vode do ostvarenja cilja [2]. Akter može biti čovek, drugi računar ili čak i vreme.

Slučajevi korišćenja mogu biti prikazani u tekstualnom obliku ili u obliku dijagrama. Osnovni slučajevi korišćenja *CodeBlock* komponente su: kreiranje *CodeBlock* komponente, dodavanje ulaznih terminala, brisanje ulaznih terminala, izmena ulaznih terminala, dodavanje izlaznih terminala, brisanje izlaznih terminala, izmena izlaznih terminala, unos izvornog koda komponente, izmena izvornog koda komponente, generisanje koda, i brisanje *CodeBlock* komponente.

Na slici 1 prikazan je dijalog za konfiguraciju *CodeBlock* komponente. Ulazni i izlazni terminali se definišu unutar *inputs* i *outputs* tabela, respektivno. Prilikom definisanja terminala, potrebno je zadati njegov naziv i tip. Terminal može biti tipa *real, int* ili *unsigned_int*. *Code* blok sa opisom funkcionalnosti se unosi u veliko tekstualno polje.

Na slici 2 prikazane su dve *CodeBlock* komponente, pri čemu komponenta *CodeBlock1* poseduje dva ulazna i dva izlazna terminala, a komponenta *CodeBlock*2 po jedan ulazni i izlazni terminal.



Slika 1: *Dijalog za konfiguraciju CodeBlock komponente*

2.2. *Generisanje šablona*

DSP komponente koje je korisnik naveo na dijagramu će se u procesu kompajliranja šeme prevesti na C programski kod upotrebom šablona. Šablon predstavlja “recept” za kreiranje komponente i obezbeđuje da svaka komponenta ima istu strukturu. Međutim, iako komponente imaju istu strukturu, to ne znači se ponašaju isto. Njihovo ponašanje zavisi od vrednosti zadanih promenljivih, te funkcije kojom je opisano njihovo ponašanje. Za većinu komponenti unutar sistema, šabloni su unapred definisani, jer se opis komponente unapred zna. Međutim, zbog činjenice da opis CodeBlock komponente nije unapred poznat, nego zavisi od podataka unesenih od strane korisnika prilikom kreiranja komponente, šablon je potrebno kreirati dinamički.

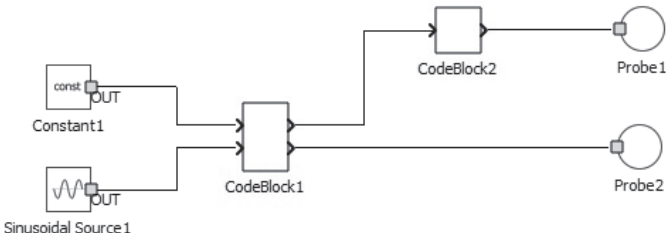
Struktura šablona za opis svake komponente je sačinjena od sledećih delova: definisanje globalnih promenljivih, definisanje lokalnih promenljivih, definisanje inicijalizacije promenljivih, definisanje prototipova funkcija, definisanje funkcija, definisanje poziva funkcija, definisanje prototipova reset funkcija, definisanje reset funkcija i definisanje poziva reset funkcija.

Svaki od ovih delova direktno zavisi od: imena komponente, ulaznih i izlaznih terminala komponente i koda koji opisuje funkcionalnost komponente.

Šabloni su opisani pomoću *Jinja2*[1] obrađivača šablona. Obrađivač šablona je softverska komponenta koja kombinuje šablone sa modelom podataka kako bi proizvela jedan ili više

1 Jinja2, http://jinja.pocoo.org

izlaznih dokumenata. Šablon svake komponente sastavljen je od više delova, predstavljenih pomoću makroa. Makroi unutar *Jinja2* se mogu uporediti sa funkcijama u regularnim programskim jezicima. Koriste se da se često korišćeni idiomi stave unutar funkcija koje su ponovo iskoristive [3]. *Jinja2* nudi mogućnost *importa* makroa u određeni šablon, ukoliko je makro definisan u nekom drugom šablonu.



Slika 2: *CodeBlock komponente na šemi*

Čitava logika za kreiranje šablona, smeštena je u klasi *TemplateCreator*. Neke od najvažnijih metoda su:

- *macro_global_vars* – metoda na osnovu prosleđenog *dictionary*-ja statičkih varijabli, kreira istoimeni makro unutar šablona,
- *macro_local_vars* – metoda na osnovu prosleđenog *dictionary*-ja izlaznih terminala unutar šablona kreira istoimeni makro,
- *macro_function* – metoda kreira makro za generisanje funkcije, na osnovu ulaznih terminala, izlaznih terminala i koda.
- *remove_statics_from_code_block* – metoda iz kod bloka komponente izbacuje deklaraciju svih statičkih komponenti, jer je njihova deklaracija odrađena u makrou *macro_local_vars*, te ukoliko je statička varijabla inicijalizovana, tu varijablu ubacuje u listu za inicijalizaciju, koja se prosleđuje metodi *macro_init_calls,*
- *make_template* – metoda koja vrši lančano pozivanje navedenih metoda, te kao rezultat vraća string koji predstavlja izgenerisani šablon za CodeBlock komponentu.

Primer izgenerisanog šablona za *CodeBlock* komponentu sa nazivom *CodeBlock1*, ulaznim terminalom: *real in1*, izlaznim terminalima *real out1* i *real out2*, te kod blokom prikazanim na listingu 1, dat je na listingu 2.

```
static real x = 5;

if (in1 < 1) {
     int y = in1 * x;

     *out1 = abs(y);
     *out2 =  y;
}else{
     int y = 1.34 * x;
     *out1 = y * 5;
     *out2 = sqrt(y);
}
```

Listing 1. *Kod koji opisuje funkcionalnost CodeBlock komponente*

Na osnovu izlaznih terminala, kreira se makro za generisanje lokalnih promenljivih – *macro_local_vars.* Naziv lokalne promenljive se dobija tako što se na naziv terminala, kao prefiks, doda naziv komponente napisan malim slovima, ispred kog sledi donja crtica, da bi se izbeglo dupliranje imena varijabli u okviru procedure gde se te varijable koriste. Tako, ukoliko imamo izlazni terminal *term1* tipa *int* unutar komponente pod nazivom *Adder*, biće deklarisana sledeća varijabla: *int _adder_term1*.

```
{%- macro_global_vars(d)-%}
{{real}} {{d.component_name}}_x;
{%- endmacro -%}

{%- macro_local_vars(d)-%}
{{real}} {{d.component_name}}_out1;
{{real}} {{d.component_name}}_out2;
{%- endmacro -%}

{%- macro_functions(d)-%}
static inline void {{d.component_name}} ({{real}}
in1, {{real}} *out1, {{real}} *out2){
        if (in < 1) {
                int y = in1 * _codeblock1_x;

                *out1 = abs(y);
                *out2 =  y;
        }else{
                int y = 1.34 * _codeblock1_;
                *out1 = y * 5;
                *out2 = sqrt(y);
        }
}
{%- endmacro -%}

{%- macro_reset_functions(d)-%}
static inline void rst_{{d.component_name}} (){
     {{d.component_name}}_x = {{cast_if_nee-
ded(real, real)}} 5;
}
{%- endmacro -%}
```

Listing 2. *Deo izgenerisanog šablona*

Makroi *macro_global_vars, macro_init_calls, macro_reset_prototypes* i *macro_reset_functions* se kreiraju na osnovu globalnih promenljivih, koje se definišu na osnovu statičkih promenljivih unutar koda komponente. Ime globalne promenljive prilikom generisanja se dobija na identičan način kao što je opisano kod lokalnih promenljivih, pri čemu je globalna promenjiva istog tipa kao i statička promenjiva od koje je nastala. Pomoću makroa *macro_init_calls, macro_reset_prototypes* i *macro_reset_functions* se generiše C kod koji omogućava da globalna promenjiva zadrži svoje stanje između višestrukih poziva funkcije komponente.

Makro za generisanje prototipova funkcija – *macro_prototypes* – se kreira na osnovu ulaznih i izlaznih terminala, koji predstavljaju parametre funkcije, i imena komponente na osnovu kog prototip funkcije dobija ime.

Makro za generisanje funkcija – *macro_functions* – kreira se na osnovu ulaznih i izlaznih terminala i koda komponente. Generisane funkcije nemaju povratnu vrednost, pa se povratne vrednosti smeštaju u pokazivače koji se kreiraju na osnovu izlaznih terminala.

3. GRAMATIKA C JEZIKA CODEBLOCK KOMPONENTE

Da bi se zadržala nezavisnost od platforme, odnosno procesora, funkcionalnost *CodeBlock* komponente se opisuje modifikovanim podskupom C jezika. Jezik je opisan pomoću PEG *(Parsing Grammar Expresion)*[8] gramatike. Sama gramatika nije konstruisana od nule, nego je izvršena konverzija nezavisne od konteksta[1] u PEG gramatiku. Za implementaciju gramatike i parsiranje koda, korišćen je a*rpeggio* [7], koji je kao slobodan softver dostupan na adresi: https://github.com/igordejanovic/arpeggio.

Arpeggio je interpreter PEG gramatika, implementiran kao rekurzivni silazni parser koji koristi tehnike memoizacije i *backtracking-a.* Ovakav tip parsera se naziva Pakrat parser.

Arpeggio je LL parser, koji ulazni string parsira sa leve na desnu stranu, zbog čega ne podržava levu rekurziju. S toga, prilikom implementacije PEG gramatike, odnosno konvertovanja CFG *(Context-Free Grammars)* u PEG, bilo je potrebno eliminisati levu rekurziju u svim pravilima gde se ona pojavljivala. Eliminacija je odrađena na sledeći način: Levo rekurzivno pravilo "A → Aa | B", postaje " A → B a*". Što znači, da bi se izbegla leva rekurzija, potrebno je da se prvo izdvoje sve alternative koje nisu levo rekurzivne i da se navedu upotrebom PEG sekvence. Zatim, za alternative koje jesu levo rekurzivne, navodi se samo ostatak koji nije levo rekurzivan označen operatorom '*', odnosno "nula ili više". Tako pravilo:

```
additiveExpr
   : multiplicativeExpr
   | additiveExpr '+' multiplicativeExpr
   | additiveExpr '-' multiplicativeExpr
   ;
```

postaje:

```
additiveExpr ← multiplicativeExpr / (['+', '-'], multiplicativeExpr)*;
```

1 https://github.com/antlr/grammars-v4/blob/master/c/C.g4

Arpeggio se inicijalizuje iz opisa gramatike jezika koja može biti zadata u tekstualnom obliku ili u obliku predstavljenim konstrukcijama *Python* programskog jezika. Gramatika za *CodeBlock* komponentu je opisana programskim jezikom *Python*, pri čemu je svako pravilo parsiranja opisano Python funkcijom, dok su operatori uređenog izbora i sekvence opisani listom i tipom uređene n-torke, respektivno. Ostali PEG operatori, implementirani su kao Python klase *ZeroOrMore*, *OneOrMore*, i sl. [6]

Razlika između ova dva načina inicijalizacije je jedino u brzini, s obzirom da je gramatiku zadanu u tekstualnom obliku potrebno parsirati, dok za gramatiku napisanu korišćenjem *Python*-a to nije potrebno.

Parser je opisan klasom *PythonParser*. Konstruktoru te klase se kao argumenti prosleđuju početna pravila parsiranja, kao i dodatni keyword argumenti, poput *reduce_tree* ili *debug*.

Početna pravila prosleđena parseru su *compilation_unit* i *comment*. Pravilo *comment* je prikazano u listingu 4.1, dok je pravilo *compilation_unit* prikazano na listingu 3.

```
def comment(): return [_("//.*"), _("/\*[^*]*\*/")]
```

Listing 3. *Pravilo comment*

```
//This is line comment
int state_1 = 0;
/*
This is multiline comment.
Var bellow presents some kind of flag
*/
int flag = 1;
if (flag == 0){
     state_1 = 1; //This is in-line comment
}
```

Listing 4. *Vrste komentara podržane od strane gramatike*

Pomoću pravila za komentare, u gramatiku je uključena i podrška za jednolinijske, višelinijske, kao *in-line* komentare, koji su prikazani u listingu 4.

Pravilo *compilation_unit* je početno pravilo za parsiranje svih ostalih vrsta izraza. Navedeno pravilo može da sadrži nula ili više izraza predstavljenih pravilom *translation_unit* i uvek mora da završi pravilom *EndOfFile*, koje predstavlja kraj ulaza koji se parsira. Pravilo je prikazano na listingu 5.

Pravilo *translation_unit* predstavlja trenutni ulaz koji se parsira, te se sastoji od najmanje jednog *external_declaration* pravila, koje je zapravo uređeni izbor između pravila *declaration* i *statement*, koja omogućavaju deklaraciju promenjivih, te pisanje raznih vrsta izraza. Pravila *translation_unit* i *external_declaration* su prikazana na listingu 6.

```
def compilation_unit():          return ZeroOrMore(translation_unit), EOF
```

Listing 5. *Pravilo compilation_unit*

```
def external_declaration():      return [declaration, statement]
def translation_unit():          return external_declaration,ZeroOrMore(external_declaration)
```

Listing 6. *Pravila translation_unit i external_declaration*

```
def type_qualifier():                    return 'volatile'
def type_specifier():                    return ['int', 'unsigned_int', 'real']
def storage_class_specifier():           return 'static'
def declaration_list():                  return declaration, ZeroOrMore(declaration)
def declaration_specifier():             return [storage_class_specifier,type_specifier, type_qualifier]
def declaration_specifiers():            return OneOrMore(declaration_specifier)
def declaration():                       return declaration_specifiers,Optional(init_declarator_list), ';'
```

Listing 7. *Pravilo declaration i pravila vezana specifikatore*

```
def declarator():                        return Optional(pointer), direct_declarator
def init_declarator():                   return [(declarator, '=', initializer),declarator]
def init_declarator_list():              return init_declarator, ZeroOrMore(',', init_declarator)
```

Listing 8. *Pravila init_declarator_list, init_delarator i declarator*

Kao što je poznato, prilikom deklaracije promenjive unutar C jezika, omogućeno je da se promenjivoj dodeljuju i određeni specifikatori. To su specifikatori koji ukazuju na tip promenjive, način na koji se vrši njeno skladištenje u memoriji i sl. Deo tih specifikatora je podržan i unutar implementirane gramatike pomoću pravila *declaration_specifiers*, *declaration_specifier*, *storage_class_specifier*, *type_specifier* i *type_qualifier*. Pravilo *declaration*, kao i pravila vezana za specifikatore, je prikazano na listingu 7.

Kao što se može videti na listingu 7, unutar pravila *declaration*, nakon odabira određenog specifikatora, sledi opciono pravilo *init_declarator_list* koje omogućava da se promenjiva inicijalizuje prilikom njene deklaracije. Pravilo *init_declarator_list* se sastoji od jednog ili više *init_declarator* pravila razdvojenih zarezom, što omogućava da se u jednoj liniji koda deklariše više promenjivih istog tipa i istog specifikatora. Pravila *init_declarator_list* i *init_delarator*, kao i pravilo *declarator* su prikazani na listingu 7.

Pravilo *deklarator* predstavlja sekvencu od dva pravila, pri čemu prvo pravilo omogućava deklaraciju pokazivača i ono je opciono, dok je drugo pravilo *direct_declarator*, koje omogućava kompleksnije deklaracije promenljivih poput onih prikazanih na listingu 9. Pravilo *declarator* je prikazano na listingu 8.

```
int a, b, c;
int d, e = 1;
real x, z;
unsigned_int x = y = z = 5;
```

Listing 9. *Primer deklaracija promenjivih*

Iskazi unutar gramatike su opisani pravilom *statement*, čija je implementacija prikazana na listingu 10. Pravilo *statement* je sačinjeno od uređenog izbora između pravila koja opisuju petlje, skokove, uslovna grananja i sl., kao i kombinovanje svih tih iskaza u jedan.

Prvo pravilo unutar uređenog izbora je pravilo *jump_statement* koje opisuje skokove *break* i *continue* koji su navedeni unutar izbora, iza kog obavezno sledi znak ";". Oba skoka se koriste isključivo unutar petlji, pri čemu *break* služi da se petlja napusti, dok *continue* vrši prelazak na sledeći element iteracije. Pravilo *jump_statement* je prikazano na listingu 11.

Nakon pravila *jump_statement*, sledeće pravilo je *expression_statement*, koje služi za opis izraza koji su detaljnije opisani u daljem tekstu. Pravilo *expression_stement* je prikazano na listingu 12.

Sledeće pravilo unutar uređenog izbora je pravilo *labeled_statement*, koje služi za opis dela *switch-case* uslovnog grananja. Pravilo l*abeled_statement* opisuje *case* i *default* slučajeve unutar navedenog grananja. Pravilo je prikazano na listingu 13.

Pravilo *selection_statement* opisuje razne vrste uslovnog grananja kao što su *if-else* ili *switch-case*. Pravilo je prikazano na listingu 14.

Pravilo *iteration_statement* služi za opis standardnih petlji kao što su *for*, *while* i *do-while* petlja. Pravilo je prikazano na listingu 15.

```
def statement():                         return [jump_statement, expression_statement, labeled_statement,
                                         selection_statement, iteration_statement, compound_statement]
```

Listing 10. *Pravilo statement*

```
def jump_statement():                    return ['continue', 'break'], ';'
```

Listing 11. *Pravilo jump_statement*

```
def expression_statement():              return Optional(expression), ';'
```

Listing 12. *Pravilo expression_statement*

```
def labeled_statement():                 return [(Not('default'), IDENTIFIER, ':', statement), ('case',
                                         constant_expression, ':', statement), ('default', ':', statement)]
```

Listing 13. *Pravilo labeled_statement*

```
def selection_statement():               return [('if', '(', expression, ')', statement, Optional('else',
                                         statement)), ('switch', '(', expression, ')', statement)]
```

Listing 14. *Pravilo selection_statement*

```
def iteration_statement():              return [('while', '(', expression, ')',statement),
                                                ('do', statement, 'while', '(', expression, ')', ';'),
                                                ('for', '(', Optional(expression), ';',
                                                Optional(expression), ';',
                                                Optional(expression), ')', statement),
                                                ('for', '(', declaration, Optional(expression), ';',
                                                Optional(expression), ')', statement)]
```

Listing 15. *Pravilo iteration_statement*

```
def block_item():                       return [declaration, statement]
def block_item_list():                  return block_item, ZeroOrMore(block_item)
def compound_statement():               return '{', Optional(block_item_list), '}'
```

Listing 16. *Pravila za konstrukciju blokova naredbi*

```
def postfix_expression():               return primary_expression,\
                                        ZeroOrMore(['++', '--', ('[', expression, ']'),
                                        ('.', IDENTIFIER), ('->', IDENTIFIER)])
def unary_expression():                 return [(UNARY_OPERATOR, unary_expression),
                                        ('sizeof', '(', type_specifier, ')'),
                                        ('sizeof', unary_expression),
                                        postfix_expression]
def assignment_expression():            return [(Not('('), unary_expression,
                                        Not('=='), ASSIGNMENT_OPERATOR,
                                        assignment_expression),
                                        conditional_expression]
def expression():                       return assignment_expression,\
                                        ZeroOrMore(',', assignment_expression)
```

Listing 17. *Pravila expression, assignment_expression, unary_expression i postfix_expression.*

Pravilo *compound_statement*, koje omogućava kreiranje „blokova“ naredbi unutar tela drugog iskaza. Kreiranje takvih blokova naredbi opisano je pomoću pravila *block_item_list* i *block_item*. Pravila *compound_statement*, *block_item_list* i *block_item*, navedena su u listingu 16.

Izrazi unutar gramatike su opisani pravilom *expression*, čija je implementacija prikazana na listingu 17. Pravilo *expression* sastavljeno je od jednog ili više *assignment_expressio*n pravila razdvojenih zarezom. Pravilo *assignment_expression* opisuje dodelu vrednosti desnog operanda levom operandu. Levi operand prilikom dodele vrednosti je unarni izraz, opisan pravilom *unary_expression*, dok je desni operand uslovni izraz opisan pravilom *conditional_expression*. Unarni izraz sadrži samo jedan operand i jedan unarni operator.

Rezultat parsiranja ulaza je stablo parsiranja koje se koristi u IDE okruženjima za navigaciju, bojenje sintakse i sl. jer poseduje detaljne informacije o strukturi ulaznog teksta [7]. Ukoliko prilikom parsiranja dođe do greške, parser će vratiti izuzetak.

Semantička analiza stabla parsiranja se vrši pomoću semantičkih akcija. Semantičke akcije su objekti koji se pridružuju čvorovima modela parsera i definišu akcije koje je potrebno izvesti nad čvorovima stabla parsiranja da bi se izvršila transformacija na čvorove grafa apstraktne semantike [7]. Semantička analiza se obavlja od dna ka vrhu, te se pri tome izvršavaju semantičke akcije. Stablo se obilazi u dva prolaza, i za svaki od prolaza je definisana po jedna metoda. Za prvi prolaz to je metoda *first_pass*, koja je obavezna za implementaciju. Za drugi prolaz, koristi se metoda *second_pass*, čija implementacija nije neophodna.

U gramatici su definisane dve semantičke akcije: *MathFunctionSem* i *IdentifierSem*. Semantička akcija *MathFunctionSem* proverava da li je pozvana neka od funkcija iz *math.h* biblioteke, te da li joj je prosleđen tačan broj parametara. Ukoliko se poziva funkcija koja nije iz navedene biblioteke ili ukoliko nekoj funkciji iz *math.h* biblioteke nije zadan tačan broj parametara koji prima, *first_pass* će vratiti izuzetak.

Semantička akcija *IdentifierSem* proverava da li je promenjiva koja se koristi u određenom iskazu prethodno deklarisana. Ukoliko nije, metoda *second_pass* će vratiti izuzetak sa porukom o grešci. Ova provera je implementirana u drugom prolasku kroz stablo, jer je tada stablo parsiranja kompletirano, te se lako može proveriti da je li neki od terminala stabla zapravo promenjiva, te da li se njena deklaracija nalazi negde prethodno u stablu. Upravo zbog činjenice da je stablo kompletirano tek posle prvog prolaza, ovu proveru nije moguće implementirati u metodi *first_pass*.

4. ZAKLJUČAK

Povećanje proizvodnje električne energije izazvalo je potrebu za većom efikasnošću njene proizvodnje i distribucije. Ova efikasnost je znatno uvećana zahvaljujući brzom razvoju energetske elektronike i tehnologije digitalne obrade signala u nekoliko poslednjih decenija, kao i testiranjem energetskih sistema na posebnim uređajima, gde se realni sistem zamenjuje modelom. Zbog širokog dijapazona primene tehnologije obrade digitalnih signala, nije moguće odrediti konačan skup komponenti koje bi određenom korisniku mogle biti potrebne. U ovom radu je opisana generička *CodeBlock* komponenta koja

omogućava korisnicima da kreiraju svoje DSP komponente u sklopu editora šema *TyphoonHIL* kompanije.

Najkompleksniji korak pri implementaciji ovog rešenja je bila implementacija PEG (*Parsing Expression Grammar*) gramatike kojom je opisana modifikovana verzija jezika C koja je korišćena prilikom opisa funkcionalnosti *CodeBlock* komponente. Implementacijom ove gramatike postignuto je da kompanija *TyphoonHIL* ima potpunu kontrolu nad time koje jezičke konstrukcije dozvoljava ili ne dozvoljava prilikom opisa funkcionalnosti *CodeBlock* komponente. Gramatika je opisana jezikom *Python*, te služi za inicijalizaciju *arpeggio* parsera, uz pomoć kojeg se vrši parsiranje ulaznog koda, kao i razne semantičke provere. Zatim, usledila je implementacija generatora šablona koja se direktno oslanja na parsiranje. Kreirani šabloni su opisani *Jinja2* jezikom za opis šablona.

Upotrebom generičke CodeBlock komponente ubrzan je razvoj specifičnih DSP komponenti jer razvoj ne obavljaju programeri već krajnji korisnici sistema. Takođe, ovim pristupom omogućeno je da se od strane Typhoon alata za modelovanje ponudi skup najčešće i najšire korišćenih DSP komponenti a da se pri tom izbegne ograničavanje korisnika u slučaju postojanja specifičnih zahteva.

5. LITERATURA

[1] Steven W. Smith, Digital Signal Processing: A Practical Guide for Engineers and Scientists, Newnes, Newnes, New South Wales (2002)

[2] Alistair Cockburn ,*Writting effective use cases,* Addison-Wesley Professional, Boston (2000)

[3] Jinja 2 templates, http://jinja.pocoo.org/docs/templates/

[4] Definition of grammar in English, http://www.oxforddictionaries.com/definition/english/grammar

[5] Formal Grammars and Languages, http://www.cs.ucr.edu/~jiang/cs215/tao-new.pdf

[6] Context-Free Grammars, https://www.cs.rochester.edu/~nelson/courses/csc_173/grammars/cfg.html

[7] Igor Dejanović, Branko Perišić, Gordana Milosavljević, Arpeggio: Pakrat Parser Interpreter, YuInfo Kopaonik, Srbija (2010)

[8] Bryan Ford, *Parsing Expression Grammars: A Recognition-Based Syntactic Foundation*, POPL, Venice(2004)

Alen Suljkanović, master inženjer elektrotehnike i računarstva, TyphoonHil, Novi Sad.
Kontakt: alen.suljkanovic@typhoon-hil.com
Oblast interesovanja: Modelima upravljan razvoj softvera, Agilne metodologije, Jezici specifični za domen

Dr Gordana Milosavljević, Fakultet tehničkih nauka, Univerzitet u Novom Sadu.
Kontakt: grist@uns.ac.rs
Oblast interesovanja: Modelima upravljan razvoj softvera, Agilne metodologije, Razvoj informacionih sistema vođen modelima

Dr Igor Dejanović, Fakultet tehničkih nauka, Univerzitet u Novom Sadu.
Kontakt: igord@uns.ac.rs
Oblast interesovanja: Jezici specifični za domen, Modelima upravljan razvoj softvera, Upravljanje konfiguracijom softvera

Dr Dušan Majstorović, TyphoonHil, Novi Sad
Kontakt: dusan@typhoon-hil.com
Oblast interesovanja: Projektovanje i verifikacija digitalnih sistema baziranih na programibilnim sekvencijalnim mrežama (FPGA), Računarski sistemi visokih performansi