METODĂ DE DEZVOLTARE A SISTEMELOR ÎNCORPORATE PRIN CONFIGURARE ȘI GENERARE COD BAZATE PE METAMODELE JSON

DOI: 10.5281/zenodo.4269373
CZU: 004.27/.004.4

Doctorand **Andrei BRAGARENCO**
E-mail: andrei.bragarenco@mib.utm.md
Universitatea Tehnică a Moldovei

METHOD FOR EMBEDDED SYSTEMS DEVELOPMENT BY CONFIGURATIONS AND CODE GENERATION BASED ON JSON METAMODELS

Summary. The development of microelectronic technologies provides more and more solutions for System on Chip with increasing processing power, which has allowed to multiply in an increased tempo the number of embedded devices. A great advantage is brought by the project development platforms through configuration and code generation, involving the reduction of the effort to adapt the reusable resources within the project and their formal configuration. Following the use of such automation technologies, the development effort is focused to meet the high-level functional requirements and application configuration using of formal or metalanguage methods. In this paper is proposed a method of developing embedded systems by configuration and code generation based on JSON metamodels coming at the end with a practical realization of such a system.

Keywords: embedded systems, code generator, architecture, metamodel, automation, design.

Rezumat. Dezvoltarea tehnologiilor microelectronice pune la dispoziție tot mai multe soluții de sisteme într-un cip cu putere de procesare în creștere, fapt care a permis multiplicarea într-un ritm sporit a numărului de dispozitive încorporate. Un mare avantaj îl oferă platformele de dezvoltare a proiectelor prin configurare și generare cod, implicând reducerea efortului de adaptare a resurselor reutilizabile și configurarea formală a acestora. În urma utilizării tehnologiilor de automatizare, sunt satisfăcute cerințele funcționale de nivel înalt ale aplicației, prin utilizarea de metode formale sau metalimbaje. În aceasta lucrare este propusă o metodă de dezvoltare a sistemelor încorporate prin configurare și generare cod bazate pe metamodele JSON. Se prezintă o realizare practică a unui asemenea sistem.

Cuvinte-cheie: sisteme încorporate, generator cod, arhitectură, metamodel, automatizare, proiectare.

INTRODUCERE

În prezent, sistemele încorporate își cresc continuu complexitatea hardware și software, optând pentru soluții cu un singur cip, System-on-Chip (SoC). Totodată, nevoile de piață ale proiectelor SoC sporesc rapid și pun presiuni pe producători, grăbind implementarea noilor produse. Ca urmare a acestor tendințe noi, industriile de semiconductori adoptă procese de co-proiectare hardware/software, menite să elaboreze sisteme de nivel înalt de abstractizare, exprimate prin seturi de macro-blocuri reutilizabile hardware și software [1].

Sistemele încorporate sunt utilizate în mai multe domenii, cum ar fi robotica, mașinile inteligente, rețelele de senzori fără fir, alcătuite din dispozitive minuscule încorporate. Acestea pot fi definite ca sisteme de senzori distribuiți, cu o infrastructură configurată automat. Pentru a obține fiabilitate și performanță în dezvoltarea lor, apelarea la metode formale este una ambițioasă. Metodele formale permit specificarea la un nivel înalt de abstractizare, evită ambiguitatea și oferă tehnici de verificare. Cu toate acestea, trecerea de la proiectarea de nivel înalt la implementare rămâne un pas informal și predispus la erori. Formalizarea acestei etape, precum și automatizarea ei este un domeniu atractiv de cercetare [2].

Ingineria Bazată pe Modele – Model Driven Engineering (MDE) se concentrează pe un proces de proiectare în care modelul să fie obiectul central de dezvoltare. Software-ul final livrat este derivat de la aceste modele și nu e niciodată modificat direct. Instrumentele sunt capabile să transforme modelele proiectate în implementări cu limbajele dorite. Generarea automată de cod și MDE este foarte promițătoare, contribuind la reducerea defectelor și sporirea productivității [3; 4].

O tendință importantă a domeniului Internetul Lucrurilor – Internet of Things (IoT), este Internetul a Orice – Internet of Everything, ceea ce înseamnă că orice obiect sau lucru care ne înconjoară este un potențial participant la o rețea globală. De regulă, un lucru sau obiect conectat la Internet constituie un

sistem încorporat ce presupune, adeseori, încorporarea unei unități de procesare cu performanță mică într-un obiect de uz zilnic. Întrucât numărul de lucruri conectate la Internet crește exponențial, reutilizarea soluțiilor existente este decisivă pentru a face față cererii mari de lucruri care trebuie conectate la rețea. Pe lângă reutilizarea în sine, abordarea arhitecturală corectă care să fie reutilizabilă reprezintă un factor cheie. Reutilizabilitatea presupune generalizarea unor funcționalități comune pentru a fi implicate masiv în dezvoltarea dispozitivelor încorporate și, respectiv, crearea soluțiilor în baza lor.

Un impact considerabil în optimizarea procesului de dezvoltare îl are existența instrumentului de automatizare. Ingineria automatizată de dezvoltare implică eforturi computaționale în procesul de inginerie software, în vederea automatizării parțiale sau complete a acestor activități, sporind considerabil calitatea și productivitatea. Automatizarea include studiul tehnicilor de construcție, de înțelegere, adaptare și modelare a ambelor domenii, atât a produselor software, cât și a proceselor de dezvoltare. Abordări automatizate de inginerie software au fost aplicate în multe domenii ale acesteia [5].

În lucrarea dată este propusă o metodologie de automatizare a procesului de dezvoltare a sistemelor încorporate prin limbajul de metamodele JavaScript Object Notation (JSON). Instrumentele pentru crearea limbajelor de modelare specifice domeniului, Domain-Specific Modeling Languages (DSML), devin tot mai acceptate în industria software-ului în vederea dezvoltării soluțiilor specifice pentru probleme specifice. Metamodelul de bază al instrumentelor este de o importanță crucială, deoarece servește ca bază pentru toate etapele ulterioare, cum ar fi crearea de coduri sau gestionarea programată a datelor modelului. Prin urmare, accesul la metamodel trebuie să fie foarte simplu, iar consumul de memorie cât mai mic [6].

Vizualizarea grafică a modelelor ajută la perceperea vizuală a sistemelor și facilitează perceperea în ansamblu a elementelor arhitecturale ale resurselor programabile dezvoltate. O modalitate simplă de creare și vizualizare a diagramelor model este Graphviz, un software liber. Vizualizarea modelelor cu ajutorul Graphviz este o modalitate de a reprezenta informațiile structurate prin diagrame ale graficelor și rețelelor abstracte. Are aplicații importante în rețelistică, bioinformatică, inginerie software, proiectare de baze de date și web, învățare automată, interfețe vizuale pentru alte domenii tehnice [7]. Notarea Graphviz este destul de simplă în esență, fapt pentru care a și fost utilizată de autor în validarea vizuală a rezultatelor.

MATERIALE ȘI METODE

A. Considerente arhitecturale

Conform teoriei sistemelor, un dispozitiv este considerat un ansamblu de componente care operează pentru a produce un rezultat. Acesta colectează informații din interfețele de intrare și furnizează rezultatul către interfața de ieșire. Fiecare componentă ar putea fi abstractizată ca model definit de o funcție de transfer ce evaluează semnalele de intrare și furnizează semnalele rezultate. Interfețele de intrare, precum și cele de ieșire, dețin valorile semnalelor și pot fi accesate în orice moment prin intermediul metodelor de citire a interfețelor.

În specificația AUTOSAR a driverului de intrări-ieșiri digitale, Digital Input Output (DIO), un pin IO digital de uz general reprezintă un canal DIO, iar un grup de canale este o combinație logică formală a mai multor canale DIO adiacente dintr-un port DIO [9].

Urmând practicile din specificația AUTOSAR DIO, vom introduce următoarele noțiuni:

- *Channel* – canalul care reprezintă un semnal al componentei, accesibil printr-o interfață.
- *Group* – un set de canale de semnale similare din interfață ce furnizează valori pentru același domeniu sau parametru și reprezintă o grupare logică a canalelor în cadrul unei componente.
- *Push/Pull* – metoda folosită pentru a opera cu un canal în scopul aplicării sau extragerii informațiilor din acesta. Aceeași metodă este aplicată tuturor canalelor incluse în același grup. Ea ar putea fi de tip *Push*, ceea ce însemnă că aceste canale acceptă schimbarea valorii sale printr-o metodă specifică, sau de tip *Pull*, ceea ce înseamnă că este evaluată intern și accesibilă printr-o metodă la cerere (figura 1).
- *Component* – un set de funcționalități sau metode reprezentate de funcții de transfer oferite de componentă și un ansamblu de canale ce reprezentă tipurile de interfețe disponibile.
- *Runnable* – sarcină internă care definește comportamentul componentelor. Poate utiliza accesul canalului intern sau extern prin metodele de tip *Push/Pull* atașate fiecărui grup de canale.

La un nivel aplatizat al dispozitivului, sistemul va arăta ca un ansamblu de canale ce interacționează prin funcții de transfer repartizate pe grupuri și componente (figura 2).

B. Descrierea arhitecturii prin metamodele bazate pe JSON

Arhitectura sistemului poate fi descrisa prin definirea unui metamodel utilizând formatul JavaScript Object Notation (JSON) [7]. Abordarea JSON permite determinarea metamodelelor într-un mod sim-

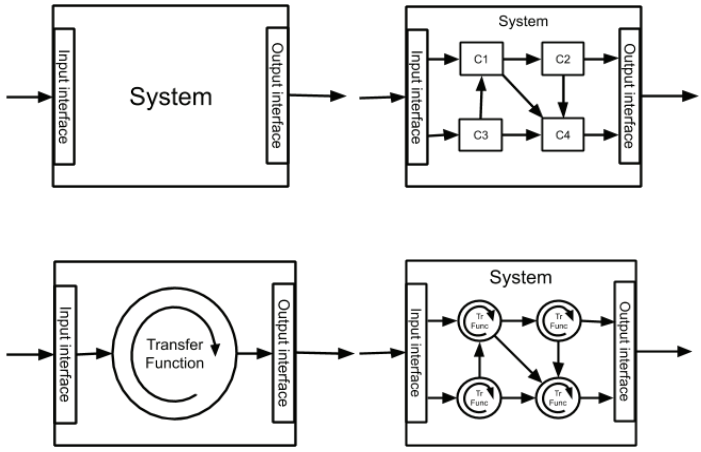

Figura 1. Canale interconectate prin metodele de tip Push și Pull.

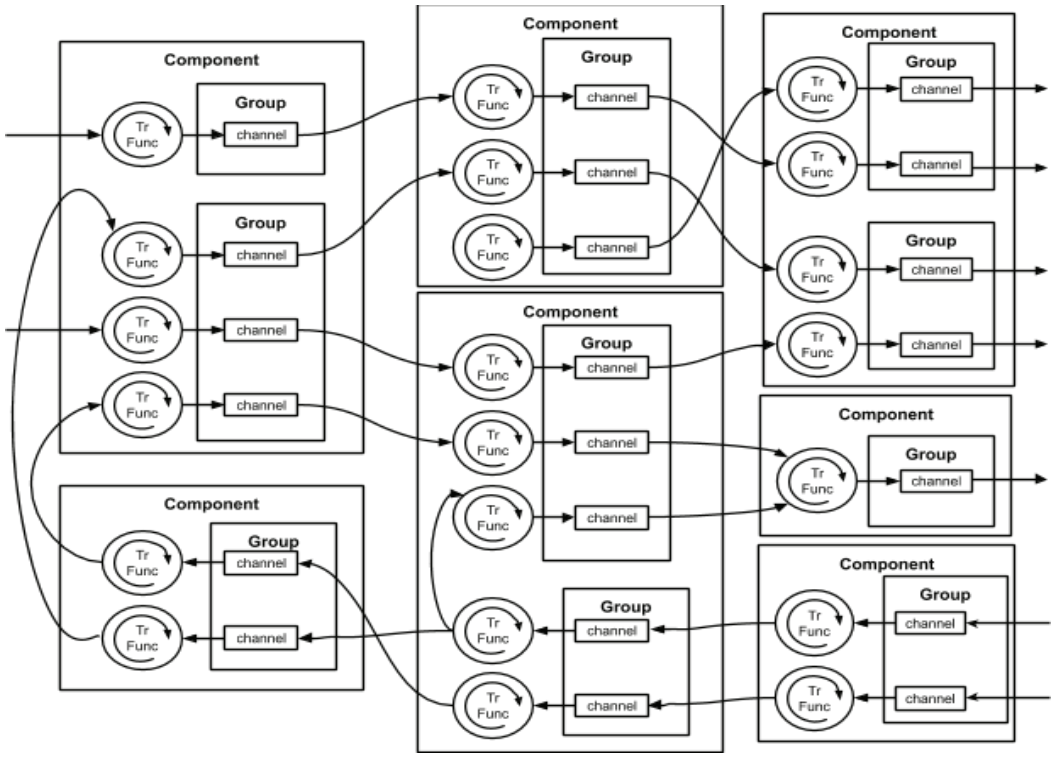

Figura 2. Vedere aplatizata a arhitecturii sistemului.

plu și clar. Formatul poate fi citit atât de utilizatorul uman, cât și suportat nativ de multe limbaje de programare care ar putea fi folosite pentru a-l analiza, așa ca Python sau JavaScript. Intenția prezentată aici este de a demonstra o metodă minimalistă, clară și ușor de gestionat, pentru determinarea arhitecturii care poate fi implementată cu eforturi minime. Conform metodei prezentate în această lucrare, vor exista trei tipuri de fișiere format JSON ale platformei: *manifest de platformă*, *definiții de componente* și *configurații de proiect*.

1. *Manifest de platformă*

Punctul de intrare al metodei de descriere este un fișier manifest în format JSON în care sunt enumerate toate componentele disponibile în platformă pentru configurarea proiectului. Fișierul manifest, *platform.JSON*, este stocat într-un repozitoriu on-line, format git și conține descrieri în formatul prezentat în exemplul de mai jos. Fișierul dat este modalitatea de informare a utilizatorului platformei despre disponibilitatea resurselor în cadrul acesteia. Așadar, oferim o mostră de fișier:

```
   {
   "Description": "Platform manifest",
   "Type": "Platform",
   "git":      "https://github.com/<file_
path>,git",
   "Path": "TOOLS/app_builder/",
```

```
   "Components": {
     "<component_name>":{
         "git":„https://github.com/<file_
path>.git",
        "Path": "MCAL/mcal_adc/"
     },
   "sensor": {
   "git":      "https://github.com/<file_
path>.git",
   "Path": "ESW/sensor/"
    }
   }
   }
```

unde:

"Description" – descrierea generală a conținutului fișierului;

"Type" – indicatorul de tip al fișierului, în cazul dat arată că fișierul conține referințe la componentele platformei.

"**Components**" – containerul tuturor referințelor componentelor disponibile în platformă;

"<component_name>" – numele componentei urmată de descrierea acesteia;

"git" – locația repozitoriului git pentru resursele componentei care conține codul sursă și alte resurse legate de componentă;

"Path" – calea recomandată de amplasare a componentei în directorul de proiect care poate fi înlocuită în configurația JSON cu cea reală în proiect.

2. *Definiția de componentă*

Fiecare componentă dezvoltată pentru utilizarea în platformă este asociată cu un fișier *definition.JSON* amplasat în directoriul principal al componentei. Definiția componentei conține constrângeri, recomandări, anumite asumări pentru configurarea componentelor proiectului. Definiția de componentă este un concept hibrid dintre manifest și definiția tipului de date. Conținutul său poate fi dedus după exemplul de mai jos:

```
   { "Type": "Definition",
     "Components":{
       "<component_name>":{
       "git":"https://<path_to_git_
file>.git",
       "Path": "<path_in_project>",
       "Groups":{
   "<group_name>": {
     "NameSpace": "<namespace_base>",
     "Multiplicity": "0-*",
     "Push": [<list_of_push_methods>],
     "Pull": [<list_of_pull_methods>],
     "Dependency": [
         <list_of_component_dependen-
cies>
          ],
     "Channels": {
     "Multiplicity": "1-*",
     "NameSpace": "<namespace_base>",
     "Names": [<mandatory_names>]
     }
    },
    "Defines": {
      <component_parameter>:
             [<purposed_value_list>]
        }
      }
    }
  }
```

unde:

"**Components**" – conține definițiile tuturor componentelor incluse în fișierul de definiție;

"<component_name>" – reprezintă numele componentei urmat de definiția acesteia;

"git" – locația repozitoriului git pentru resursele componentei care conține codul sursă și alte resurse legate de componentă;

"Path" – calea recomandată în directorul de proiect care poate fi înlocuită ulterior în configurația JSON a configurației componentelor.

"**Groups**" – conține definițiile tuturor grupurilor din componentă;

"<group_name>" – denumirea grupului ce aparține unei componente urmat de definiția acestuia. De asemenea definește tipul grupului ca referință în configurații;

"NameSpace" – definește "<namespace_base>", baza pentru generarea incrementală automată a numelor de grupuri și canale în cadrul componentei care urmează aceeași definiție. În cazul în care nu este definit, numele de grup sau canal servește drept configurație înlocuitoare a acestei setări;

"Multiplicity" – constrângeri privind multiplicitatea grupurilor sau canalelor ce urmează aceeași definiții. Valorile de minimum și maximum sunt separate prin cratimă "-" , iar "*" reprezintă infinitate. În cazul în care configurația este exprimată printr-un singur număr, aceasta arată existența unui număr specific de instanțe de grup sau canal. Când configurația nu este definită, valoarea ei va fi stabilita implicit la configurația "0-*" – pot exista de la zero pana la infinit instanțe de grup sau canal;

"Push" – o listă de metode de tip Push disponibile în cadrul componentei, care pot fi folosite în configurații pentru accesul la canalele din grupul definiției date;

"Pull" – reprezintă o listă de metode de tip Pull disponibile în cadrul componentei, care pot fi folosite în configurații pentru accesul la canalele din grupul definiției date;

"Dependency" – lista de dependențe recomandate pentru stabilirea legăturilor dintre componente. Nu este obligatorie și are ca scop sugerarea de conexiuni, totodată facilitează automatizarea descrierii configurațiilor;

"Names" – lista de nume predefinite, care vor fi selectate înainte de a genera nume de canale sau grupuri prin incrementare urmând configurația "NameSpace".

"Defines" – definiția de parametri specifici componentei la care este atașată definiția respectivă, urmată de sugestii ale valorilor parametrului;

Definițiile sunt stocate, împreuna cu componentele, în directoriul de bază al acestora. În cazul în care definiția există, aceasta este utilizată la configurarea componentei în cadrul platformei. Dacă nu, este creat un fișier nou al definiției după reguli generale, care poate fi adaptat în procesul configurării proiectului și adăugat la componentă.

3. *Configurație de proiect*

Configurația de proiect va fi descrisă într-un fișier *<project_name>.JSON* localizat în directoriul principal al proiectului. Pentru demonstrarea metodei vom urma un exemplu de interconectare a două componente (figura 3).

Conținutul fișierului de configurare va arăta ca în exemplul ce urmează:

```
{
  "Type": "Configuration",
  "Components":{
    "Comp_A": {
      "git": "https://<path_to_git_
file>.git",
      "Path": "ESW/Comp_A/",
      "Groups": {
      "Group_A": {
        "Dependency":"Comp_B"
        "Channels": {
          "cnl_A_1": "cnl_B_1",
          "cnl_A_2": "cnl_B_2",
          },
        "Pull": "GetB"
        }
      }
    },
  "Comp_B": {
    "git":  "https://<path_to_git_
file>.git",
    "Path": "ESW/Comp_B/",
    "Groups": {
      "Group_B": {
        "Channels": {
        "cnl_B_1": "<channel_link>",
        "cnl_B_2": "<channel_link>"
        },
          "Pull": "<pull_method>"
        }
      }
    }
  }
}
```

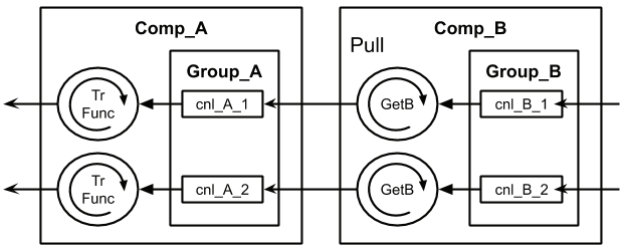

Figura 3. Exemplu de interconectare a componentelor.

unde:

"**Components**" – conține descrierea configurațiilor tuturor componentelor din proiect;

"Comp_A", "Comp_B": – denumirea componentelor din proiect urmate de descrierea configurațiilor acestora;

"git" – locația repozitoriului git pentru resursele componentei care conține codul sursă și alte resurse legate de componentă.

"Path" – calea de amplasare a componentei în directorul de proiect;

"**Groups**" – configurațiile tuturor grupurilor din componentă;

"Group_A", "Group_B" – numele de grup urmate de configurațiile acestora;

"Dependency" : "Comp_B" – componenta de care depinde grupul, respectiv, aici, canalele din grup vor fi conectate la canalele din componenta Comp_B definită ca dependență.

"**Channels**" – configurațiile tuturor canalelor din cadrul grupului;

"cnl_A_1": "cnl_B_1" – configurația care arată că canalul cnl_A_1 din grupul Group_A al componentei Comp_A este conectat la canalul cnl_B_1 din grupul Group_B al componentei Comp_B;

"Pull": „GetB" – în contextul exemplului de mai sus, configurația arată că canalul cnl_A_1 accesează datele canalului cnl_B_1 prin metoda de tip Pull GetB definită componenta Comp_B.

În exemplul prezentat este utilizat un set minim necesar pentru a descrie o configurație, dar pot exista și configurații suplimentare specifice proiectului, cum ar fi descrierea componentelor, tipurile componentelor, ale grupurilor, sau altele după necesitate.

C. Configurarea componentei din platformă.

Metoda expusă în această lucrare presupune că resursele componentei sunt compuse din trei parți – 1. funcționalitățile componentei; 2. adaptorul componentei la platformă; 3. configurația de interconectare a componentelor (figura 4).

Conform metodei date, funcționalitățile componentei pot fi definite în cadrul procesului de proiectare sau preluate ca resursă externă, celelalte două părți fiind posibil de generat în procesul de dezvoltare a proiectului utilizând conceptele platformei propuse în acesta lucrare.

Funcționalitățile componentei – un ansamblu de funcționalități, cum ar fi interfețe, funcții de transfer, modele comportamentale și altele specifice componentei date organizate în librării. Aceste funcționalități pot fi definite în cadrul procesului de dezvoltare a componentei sau preluate ca resurse terțe drept cod sursă sau librării precompilate. Funcționalitățile sunt reprezentate de către un fișier antet unde se află declarațiile acestora pentru a fi invocate. Prototipurile funcționalităților respective ar putea arăta ca în exemplele de mai jos:

```
   int ReadSpecificValue(); // în calita-
te de interfață
   int    EvaluateFunction(int    in);//
funcț. de transfer
   int DoSpecificAction(); // model com-
portamental
```

Adaptare la platformă – este partea componentei care adaptează funcționalitățile librăriei pentru a le utiliza în cadrul platformei. Aici sunt localizate structurile de date de uz general specifice platformei, cum ar fi definițiile de canale și de grupuri, metodele prin intermediul cărora acestea sunt gestionate urmând metodologia specifică platformei. Tot aici sunt indicate valorile implicite pentru parametrii componentei, folosite în cazul în care acestea nu sunt redefinite de configurația componentei.

Această abordare separă funcționalitățile de adaptorul la platformă și le face disponibile pentru a fi utilizate și în afara platformei, iar resursele terților să fie adaptate pentru utilizarea în platformă. În cazul componentelor platformei, partea de adaptare vine la pachet cu componenta, iar în cazul adaptării resurselor terțe va fi necesar un efort suplimentar de a defini partea de adaptare a resurselor la platformă.

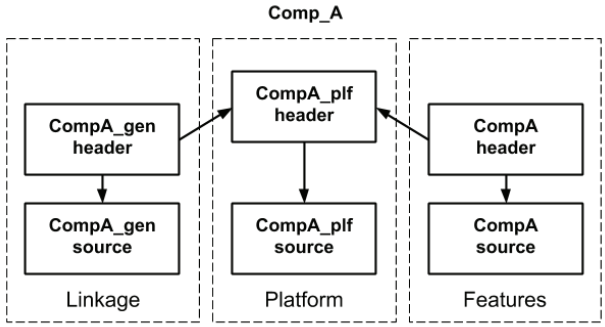

Figura 4. Infrastructura internă a componentei.

Urmând exemplul a două interconexiuni de componente, prezentate în secțiunea de definire a configurației din această lucrare, setul minim de configurații specifice platformei pentru componenta CompA va arăta după cum urmează:

pentru antetul platformei *comp_a_plf.h*:

```
   #ifndef _COMP_A_PLF_H_
   #define _COMP_A_PLF_H_

   #include „comp_a_cfg.h”

   #ifndef COMP_A_CONFIG
   enum COMP_A_Cnl_IdType {COMP_A_CHAN-
NEL_NR_OF = 0};
   enum COMP_A_Grp_IdType {COMP_A_GROUP_
NR_OF = 0};
   #endif
   typedef struct COMP_A_CnlType_t {
   Std_CnlIdType linkCnlId = 0;
   Std_PullType ExtGetValue = NULL;
   Std_PhyDataType inputValue;
   Std_PhyDataType outputValue;
   } COMP_A_CnlType;

   COMP_A_CnlType* COMP_A_GetCnlRef (…);
   Std_ReturnType COMP_A_CnlSetup(…);
   Std_ReturnType COMP_A_GroupSetup(…);
   Std_ReturnType      COMP_A_SetPullMe-
thod(…);
   Std_ReturnType  COMP_A_SetGroupPull-
Method(…);

   Std_PhyDataType  COMP_A_GetValue(Std_
CnlIdType);

   #endif
```

iar pentru implementare *comp_a_plf.c*:

```
   #include „comp_a_plf.h”

   COMP_A_CnlType    COMP_A_Cnls[COMP_A_
CHANNEL_NR_OF];
   COMP_A_CnlType*   COMP_A_Grps[COMP_A_
GROUP_NR_OF];

   COMP_A_CnlType* COMP_A_GetCnlRef (…)
{…}
   Std_ReturnType COMP_A_CnlSetup(…){…}
```

```
   Std_ReturnType   COMP_A_GroupSetup(…)
{…}
   Std_ReturnType      COMP_A_SetPullMe-
thod(…){…}
   Std_ReturnType   COMP_A_SetGroupPull-
Method(…){…}
   Std_PhyDataType COMP_A_GetValue(Std_
CnlIdType Id){
   /* exemplu a unei funcţii de tip pull
*/
      COMP_A_CnlType* cnlRef = COMP_A_
GetCnlRef(Id)
   Std_PhyDataType input, result;
   input    =    cnlRef->ExtGetValue(cn-
lRef->linkCnlId)
   result = EvaluateFunction(input);
   return result;
   }
```

Toate datele legate de canal sunt stocate în descriptori de canale definiți printr-o structură de tip **COMP_A_CnlType.** Aici sunt definite: **linkCnlId** – ID-urile canalelor din **CompB** urmând exemplul de mai sus utilizat pentru demonstrarea metodei; **ExtGetValue** – referința la metoda de tip Pull definită în **CompB** folosită pentru citirea canalului său; **inputValue** – ultima valoare colectată prin conexiunea canalului; **outputValue** – ultima valoare evaluată a canalului.

Definiția **enum COMP_A_Cnl_IdType** este utilizată pentru stabilirea ID-urilor de canal dintr-un grup, ultimul identificator fiind numărul de canale din grup, similar pentru grupuri în componentă prin definiția de **COMP_A_Grp_IdType.**

Instanțele canalelor din componentă sunt definite de **COMP_A_Cnls[COMP_A_CHANNEL_NR_OF],** care reprezintă o listă de descriptori, structuri, de canale pentru un anumit grup, aceste fiind accesibile prin ID-ul său. Instanțele grupurilor sunt definite, la rândul său, de **COMP_A_Grps[COMP_A_GROUP_NR_OF]**, reprezentând o listă de grupuri din componenta care conține referințe la instanțele grupului de canale. În mod implicit, dacă nu este aplicată nicio configurație, listele de mai sus sunt goale, de mărime zero, dar după includerea configurațiilor, acestea vor fi redefinite.

În fiecare componentă sunt definite un set de metode-funcții pentru gestionarea canalelor, cum ar fi: **COMP_A_GetCnlRef** pentru extragerea referinței la canal prin ID-ul său; **COMP_A_CnlSetup** pentru interconectarea canalelor, în exemplul nostru pentru înregistrarea ID-urilor canalului **CompB** în descriptorii canalului **CompA**; **COMP_A_GroupSetup** pentru stabilirea conexiunilor între canalele a două grupuri; **COMP_A_SetPullMethod** pentru înregistrarea metodei de componentă tip Pull în descriptorul canalului; **COMP_A_SetGroupPullMethod** pentru înregistrarea metodei de tip Pull pentru întregul grup.

Implementarea metodei **COMP_A_GetValue** va utiliza metodele din setul de funcționalități ale componentelor, precum și metode de interfață disponibile prin referințe de tip Pull, cum ar fi **ExtGetValue**, din descriptorul canalului.

Configurația componentei presupune crearea de fișiere de configurații, antet și implementare, în baza descrierilor din fișierele de configurație ale proiectului *<project_name>.JSON* unde se regăsesc configurațiile pentru toate componentele.

Urmând exemplul de referință, pentru interconectarea a două componente din aceasta lucrare, în calitate de configurație a componentei **CompA** pentru fișierul antet *comp_a_cfg.h* vom avea:

```
   #ifndef _PROJECT_CONFIG_H_
   #define _PROJECT_CONFIG_H_

   #include „platform_config.h”

   #define COMP_A_CONFIG
   enum COMP_A_Cnl_IdType {cnl_A_1, cn-
l_A_2,
   COMP_A_CHANNEL_NR_OF};
   enum COMP_A_Grp_IdType {Group_A,
   COMP_A_GROUP_NR_OF};
   #include „ESW/Comp_A/Comp_A.h”

   #define COMP_B_CONFIG
   enum COMP_B_Cnl_IdType {cnl_B_1, cn-
l_B_2,
   COMP_B_CHANNEL_NR_OF};
   enum COMP_B_Grp_IdType {Group_B,
   COMP_B_GROUP_NR_OF};
   #include „ESW/Comp_B/Comp_B.h”

   Std_ReturnType Project_config(void);
```

Acest fișier antet de configurații, fiind inclus în fișierul antet al componentei de platformă, va redefini configurațiile implicite pentru grupuri și canale care inițial au fost definite cu mărimea zero.

Pentru fișierul implementare *comp_a_cfg.c* vom avea funcția de configurare a componentelor prin stabilirea de conexiuni intre canale după cum urmează:

```
   #include „comp_a_cfg.h”

   Std_ReturnType CompA_config(void)
   {
   Serial.begin(115200);
   Serial.println(„ES   Platform   based
Project”);
```

```
   Std_ReturnType error = E_OK;
   error   +=   COMP_A_ChannelSetup(cn-
l_A_1, cnl_B_1);
   error   +=   COMP_A_ChannelSetup(cn-
l_A_2, cnl_B_2);

   error   +=   COMP_A_SetPullMethod(cn-
l_A_1, GetB);
   error   +=   COMP_A_SetPullMethod(cn-
l_A_2, GetB);
   Serial.print("GROUP_A   configured   -
Error : ");
   Serial.println(error);

   return error;
   }
```

Prin invocarea funcției **CompA_config**, în cadrul proiectului se vor stabili conexiunile între componentele **CompA** și **CompB,** prin intermediul canalelor **cnl_A_1** și **cnl_A_2** conectate respectiv cu canalele **cnl_B_1** și **cnl_B_2,** transferul de informație fiind realizat prin metoda de tip pull **GetB** pe care **CompA** o accesează prin referință din descriptorul canalului **ExtGetValue** definit ca structură de tip **COMP_A_CnlType**.

Automatizarea procesului de configurare a proiectelor presupune utilizarea programelor specializate care permit gestionarea fișierelor de manifest platformă, de definiții ale componentelor și configurațiilor acestora, excluzând intervenții la nivel de text pentru a evita erorile de editare. Utilizarea acestor resurse de programe permite ca prin intermediul interfețelor grafice să fie generate configurații aplicând metode de selecție din resursele recomandate, în baza informației colectate din multitudinea de fișiere tip JSON urmând metodologia descrisă în lucrare.

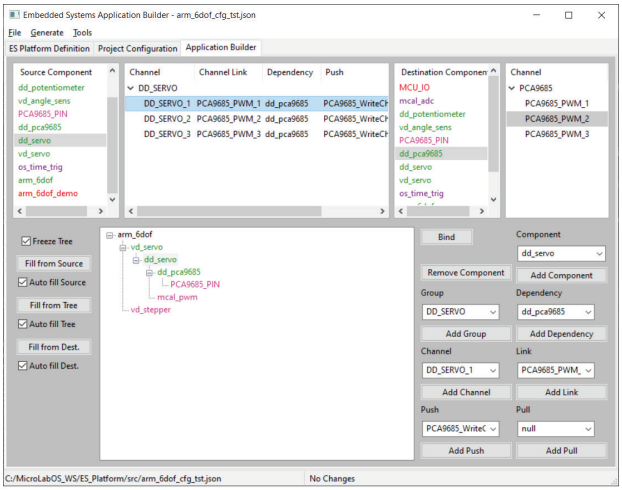

Figura 5. Gestionarea configurațiilor componentelor.

REZULTATE

În scopul demonstrării viabilității metodei prezentate în lucrarea dată a fost elaborată o resursă de program specializat în crearea și configurarea proiectelor pentru sisteme încorporate. Vederile interfeței grafice a programului sunt prezentate în figura 5.

Programul permite vizualizarea și editarea definițiilor și configurațiilor unor componente importate din cadrul unei platforme de resurse, stocate on-line și accesibile pentru realizarea proiectelor pentru dispozitive de tip sistem încorporat. În cazul disponibilității componentelor on-line acestea sunt importate, dacă nu, se creează o componentă dintr-un template care poate fi populat și pe final adăugat la platforma on-line. În ce privește automatizarea procesului de proiectare, programul are facilități de interconectare vizuală a componentelor și de generare automată a fișierelor sursă de configurare a componentei. De asemenea, programul permite vizualizarea grafică a componentelor și a interconexiunilor prin crearea diagramelor arhitecturale aplatizate, similare cu cea prezentată în figura 6, în format Graphviz [7], care verifică și confirmă corectitudinea rezultatului.

DISCUȚII

În această lucrare este propusă o metodă simplă de dezvoltare a proiectelor pentru dispozitive încorporate. Elementele principale ale metodei constituie componentele, accesibile ca resurse on-line prin intermediul serviciilor de versionare git, și metodologia de stabilire a interconexiunilor între componente. Elementele descriptive ale metodei sunt bazate pe metalimbajul JSON, el fiind unul pe înțelesul utilizatorului uman și acceptat ca nativ pentru unele limbaje de programare, ceea ce facilitează procesul de dezvoltare a instrumentelor menite să gestioneze proiectele prin metoda propusă.

Metodologia dată este dezvoltată în scopul asigurării cu instrumente de proiectare în baza concep-

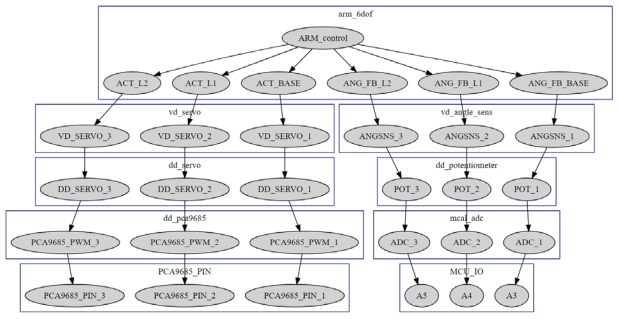

Figura 6. Vizualizare grafică a interconexiunilor între canale în format Graphviz.

tului de arhitectură pe nivele destinat aplicațiilor în sisteme din domeniul Internetul Lucrurilor (IoT), propus în lucrarea [8], unde funcționalitățile unui sistem pot fi grupate într-un set definit de componente generice, tipice pentru un dispozitiv încorporat. Aceste componente conțin funcționalități în conformitate cu problema pe care o soluționează, cum ar fi achiziția de semnal, acționarea, interacțiunea cu utilizatorul, comunicarea, stocarea, managementul energiei sau sistemele de operare.

Metodologia propusa urmează a fi dezvoltată în continuare, următorii pași fiind îndreptați spre acoperirea necesităților, după cum urmează:

- elaborarea funcționalităților de stabilire automată a conexiunilor în bază constrângerilor din definițiile componentelor și a recomandărilor bazate pe statisticile proiectelor anterioare elaborate cu ajutorul acestei platforme; inserarea automată de componente intermediare urmărind dependențele între componente și statisticile acumulate în proiectele anterioare minimizând implicarea utilizatorului;
- distribuția aplicației pe multitudinea de unități de procesare din cadrul sistemului, implicând inserarea automată a componentelor specifice comunicațiilor în rețea pentru asigurarea transferului de informații dintre dispozitivele sistemului organizate într-o rețea tip IoT;
- dezvoltarea de template-uri generice de proiecte oferite în calitate de demonstrație a componentelor platformei cu posibilitatea dezvoltării ghidate de aceste template-uri;
- dezvoltarea modalității de suplinire a colecției de resurse ale platformei prin intermediul comunității de utilizatori ai platformei.
- adăugarea elementelor de securitate cibernetică, indispensabile în sistemele din domeniul Internetului Lucrurilor.

CONCLUZII

Reutilizarea resurselor de program este esențială pentru ritmul actual de dezvoltare a tehnologiilor informaționale. Elaborarea soluțiilor bazate pe resurse reutilizabile, față de cele dezvoltate manual în cadrul proiectului, oferă numeroase avantaje, inclusiv reducerea timpului de realizare a proiectului, stabilitatea sistemului, modularitate și altele.

Un avantaj și mai mare îl oferă platformele de dezvoltare a proiectelor prin configurare și generare cod, care reduc efortul de adaptare a resurselor reutilizabile în cadrul proiectului și configurarea formală a acestora. În urma utilizării tehnologiilor de automatizare, efortul principal este direcționat pe satisfacerea cerințelor funcționale de nivel înalt ale aplicației, prin utilizare de metode formale sau metalimbaje.

În această lucrare a fost elaborată o metodă de localizare a resurselor reutilizabile prin intermediul fișierului de manifest al platformei, interconectarea după un principiu comun pentru întreaga platformă, adaptarea resurselor terțe la mecanismele de interconectare adoptate în cadrul platformei. De asemenea, s-a realizat o bază, sub forma unei platforme, pentru automatizarea procesului de proiectare în temeiul unor recomandări din fișierele de definiție a componentelor reutilizabile și a statisticilor, fapt ce oferă noi oportunități și provocări pentru dezvoltarea metodei propuse.

BIBLIOGRAFIE

1. Besana M. and Borgatti M. Application Mapping to a Hardware Platform through Automated Code Generation Targeting a RTOS: a Design Case Study, 2003 Design, Automation and Test in Europe Conference and Exhibition.

2. Houhou S., Kahloul L., Benharzallah S. and Bettira Roufaid - Framework For Wireless Sensor Networks Code Generation From Formal Specification, Computer Science & Information Technology (CS & IT) Computer Science Conference Proceedings (CSCP) Natarajan Meghanathan et al. (Eds): NeCoM, SEAS, CMCA, CSITEC – 2017, p. 35-52, 2017. DOI: 10.5121/csit.2017.71204

3. Weigert T., Weil F. van den Berg A., Dietz P., and Marth K. Automated Code Generation for Industrial-Strength Systems – 2008. In: Proceedings of the 2008 32nd Annual Ieee International Computer Software and Applications Conference.

4. Ciccozzi F., Cicchetti A., Sjödin M. Full Code Generation from UML Models for Complex Embedded Systems. In: Second International Software Technology Exchange Workshop (STEW), 2012. Publisher: Swedsoft (on-line).

5. Bousetta B., Omar El Beggar and Taoufiq Gadi. Automating Software Development Process: Analysis-PIMs toDesign-PIM Model Transformation. In: International Journal of Software Engineering and its Applications, vol. 7, no. 5 (2013), p. 167-196. DOI 10.14257/ijseia.2013.7.5.17

6. Markus G., Bayer J., Höfner J. M. and Boger M. Approach to Define Highly Scalable Metamodels Based on JSON. BigMDE@STAF, 2015.

7. Graphviz – Graph Visualization Software. [on-line] https://Graphviz.org/ (vizitat la 20.08.2020).

8. Bragarenco A., Marusic G. and Ciufurdean C. Layered Architecture Approach of the Sensor Software Component Stack for the Internet of Things. In: WSEAS Transactions on Computer Research, volume 7, 2019, Art. #15, p. 124-135.

9. AUTOSAR Release 4.2.2. Specification of DIO Driver. [on-line] https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_SWS_DIODriver.pdf (vizitat la 16.08.2020).