Marek BRYKCZYŃSKI[1]

Opiekun naukowy: Marcin SIDZINA[2]

GENEROWANIE KODU Z MODELU OPROGRAMOWANIA

Streszczenie: Przegląd i analiza wybranych narzędzi do rozwoju oprogramowania, które posiadają możliwość generowania kodu w wysokopoziomowych językach programowania. Zastosowanie techniki generowania kodu z modelu w modelu V wytwarzania oprogramowania w celu dwukierunkowego śledzenia wymagań dotyczących funkcjonalności oraz przyśpieszenia procesu wytwarzania oprogramowania.

Słowa kluczowe: wytwarzanie oprogramowania w oparciu o model, model oprogramowania, generowanie kodu, model V wytwarzania oprogramowania

GENERATING CODE FROM A SOFTWARE MODEL

Summary: Review and analysis of selected software development tools that have the ability to generate code in high-level programming languages. Application of the code generation technique from the model in the V model of software development for the purpose of bidirectional tracking of functionality requirements and acceleration of the software development process.

Keywords: model of software, code generation, model V of software development, model driven development

1. Model V wytwarzania oprogramowania

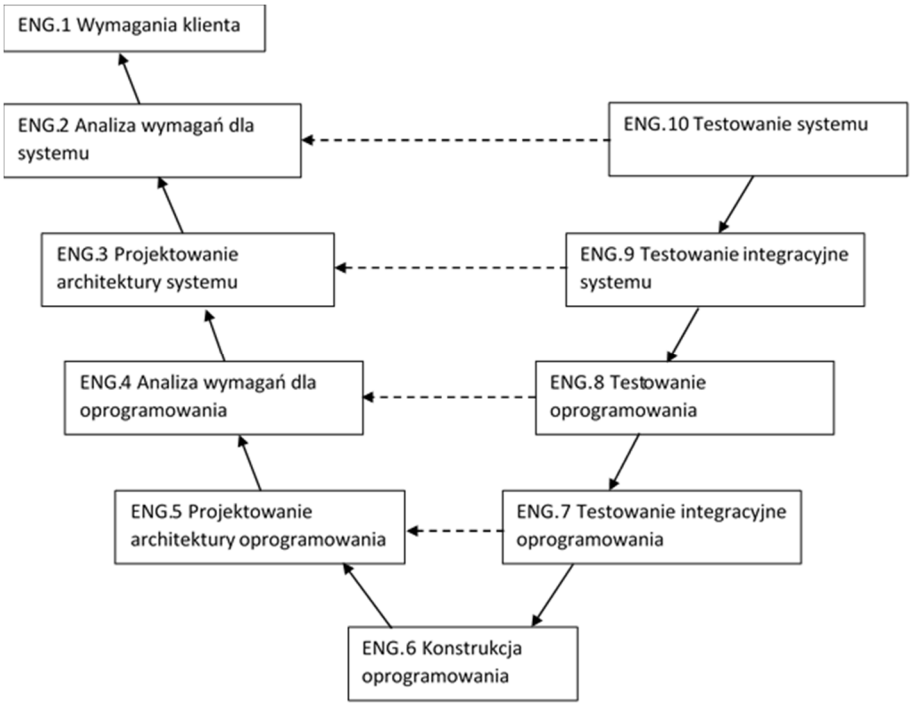
W przemyśle motoryzacyjnym oprogramowanie wbudowane dla mikroprocesorowych układów sterujących systemami w samochodach; przykładowo takich jak: komputer pokładowy, układ wspomagania kierownicy, układ poduszek powietrznych i inne; wytwarzane jest w oparciu o model V wytwarzania oprogramowania. Zastosowanie tego modelu wynika z standardów wytwarzania produktów docelowych, których częścią jest oprogramowanie. Standardami tymi są ISO 26262, który dotyczy bezpieczeństwa funkcjonalnego oraz ASPICE bazujący na

[1] Akademia Techniczno-Humanistyczna w Bielsku-Białej, Wydział Budowy Maszyn i Informatyki, Budowa i Eksploatacja Maszyn, marek.brykczynski@gmail.com
[2] dr inż., Akademia Techniczno-Humanistyczna w Bielsku-Białej, Wydział Budowy Maszyn i Informatyki, msidzina@ath.bielsko.pl

ISO/IEC 15504 (ang. Automotive Software Process Improvement and Capability Determination) tj. usprawnianie procesu wytwarzania oprogramowania w przemyśle motoryzacyjnym oraz wyznaczanie jego zdolności.

1.1. Charakterystyka modelu V wytwarzania oprogramowania

Przedstawiony na rysunku 1 model V odwołuje się poprzez znaczniki ENG.X bezpośrednio do standardu ASPICE w wersji PAM 2.5. Strzałki będące liniami ciągłymi przedstawiają kierunek przepływu danych wejściowych dla danego poziomu, strzałki kreskowane przedstawiają referencje, które są podstawą walidacji danego poziomu. Przykładowo testowanie oprogramowania waliduje czy przenalizowane i wyspecyfikowane wymagania oprogramowania są spełnione.



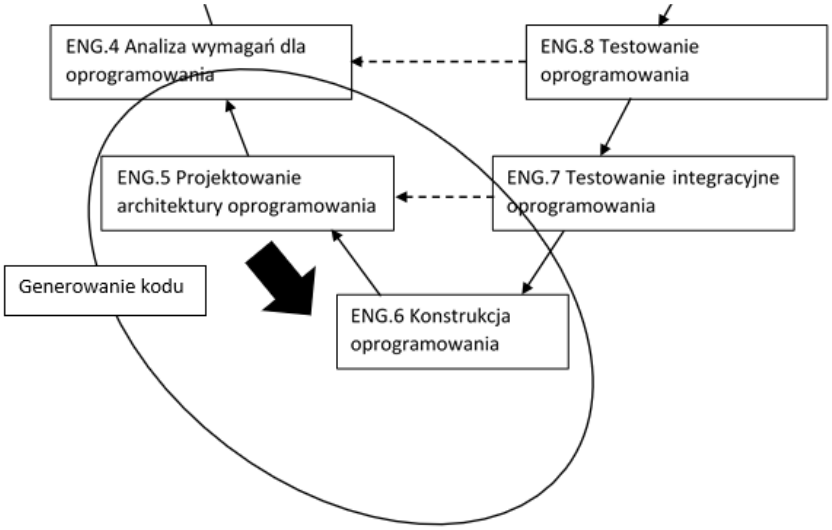
Rysunek 1. Model V wytwarzania oprogramowania – na podstawie ASPICE PAM[3] 2.5

1.2. Umiejscowienie generowania kodu w modelu V

Na podstawionego przedstawionego powyżej modelu V wytwarzania oprogramowania, model definiowany jest na poziomie projektowania architektury oprogramowania. Następnym etapem jest poziom konstrukcji oprogramowania tj. wytwarzanie i testowanie jednostkowe algorytmów napisanych przez programistę na podstawie modelu. Generując kod eliminuje się etap manualnego wytwarzania kodu

[3] Automotive Software Process Improvement and Capability Determination Process Assessment Model

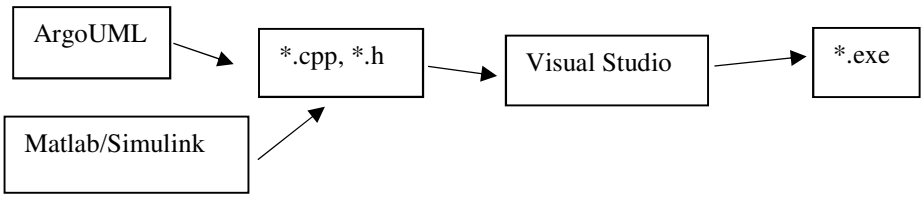
oraz dodatkowo zapewnia się automatycznie dwukierunkowe śledzenie między modelem a oprogramowaniem.



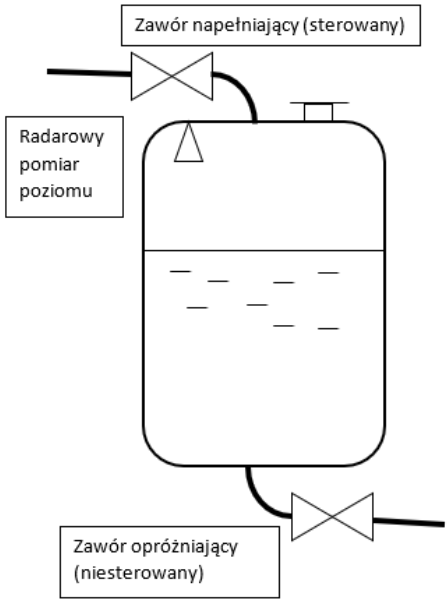
Rysunek 2. Generowanie kodu w modelu V

2. Algorytm i architektura programu do sterowania poziomem w zbiorniku

W opracowaniu tym przedstawiony zostanie uproszczony projekt architektury oprogramowania i algorytmu regulującego poziom cieczy w zbiorniku buforowym o niekontrolowanym odpływie. Architektura oprogramowania zostanie zaprojektowana w programie ArgoUML. Prosty algorytm regulacyjny zamodelowany zostanie w środowisku Matlab/Simulink. Z obydwu wspomnianych programów wygenerowany zostanie kod w języku wysokopoziomowym C++, który następnie zostanie zintegrowany i zbudowany. Docelowa aplikacja powstanie w środowisku Visual Studio jako plik wykonywalny z możliwością uruchomienia z linii komend.



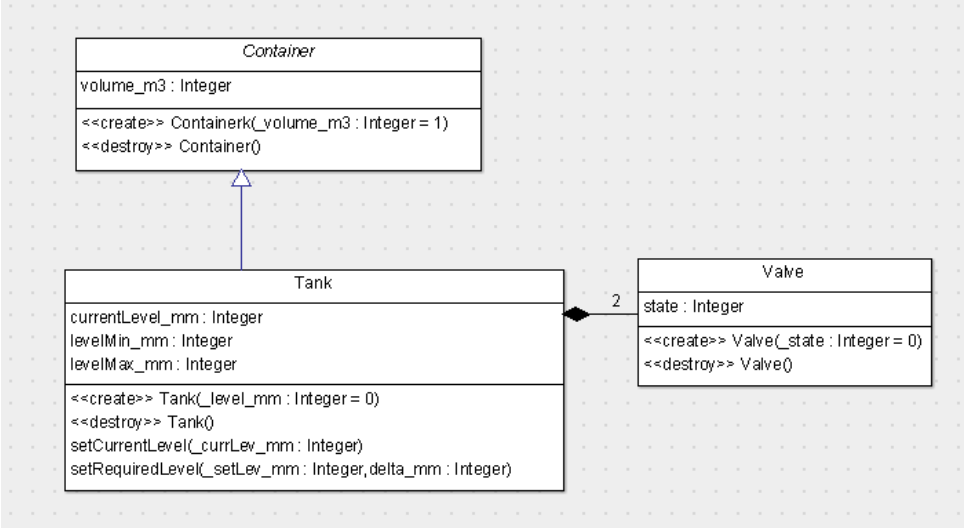
Rysunek 3. Przejście: model → wygenerowany kod → aplikacja



Obiektem automatycznej regulacji, który został wybrany do zaprojektowania architektury oraz algorytmu regulacji jest zbiornik buforowy. Do zbiornika podłączone są dwa zawory, przy czym projektowany regulator będzie posiadał możliwość sterowania tylko zaworem doprowadzającym ciecz roboczą. Zawór opróżniający sterowany jest poprzez inny regulator, nieujęty w tym opracowaniu.

Rysunek 4. Model fizycznego obiektu regulacji

2.1. Architektura w UML

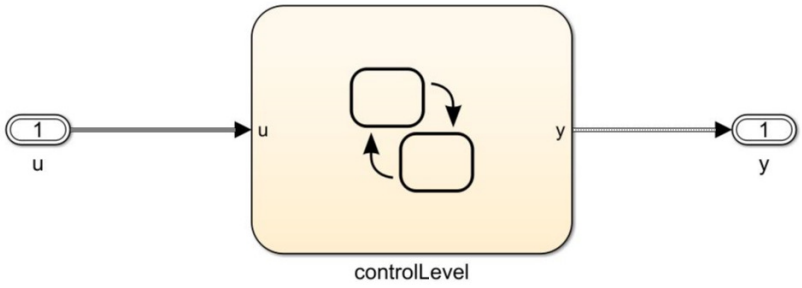


Rysunek 5. Diagram klas obiektów w układzie regulacji poziomem w zbiorniku

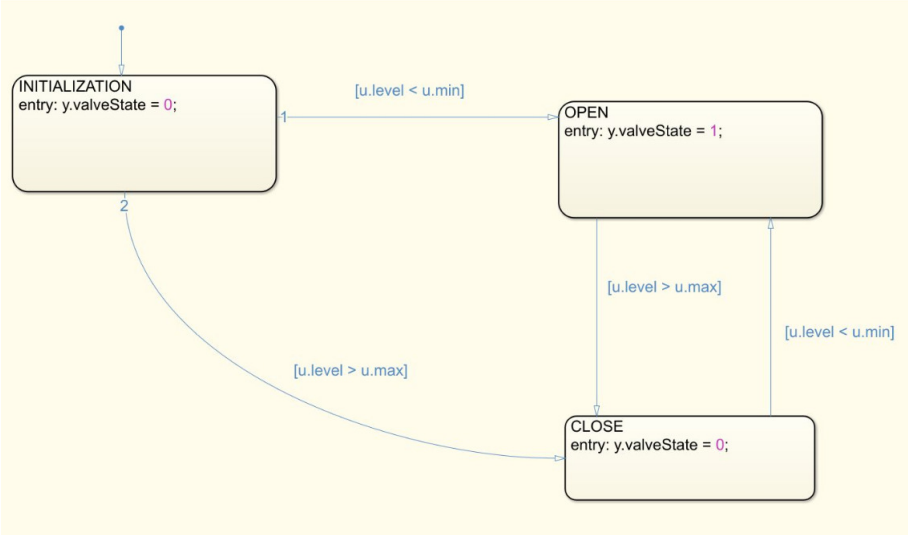
Wygenerowany kod pliku nagłówkowego klasy Tank (ang. zbiornik).

```
#ifndef Tank_h
#define Tank_h

#include "Container.h"
#include "Valve.h"

class Tank : public Container {

 public:

    Tank(Integer _level_mm = 0);
    virtual void setCurrentLevel(Integer _currLev_mm);
    virtual void setRequiredLevel(Integer _setLev_mm, Integer
delta_mm);

 public:
    Integer currentLevel_mm;
    Integer levelMin_mm;
    Integer levelMax_mm;

 public:

    /**
     * @element-type Valve
     */
    Valve myValve[ 2];
};
#endif // Tank_h
```

2.2. Model algorytmu regulacyjnego w programie Matlab/Simulink



Rysunek 6. Model symulacyjny układu regulacji



Rysunek 7. Diagram maszyny stanu realizujący algorytm regulacji

Wygenerowany kod maszyny stanu:

```
//
// File: controlLevel.cpp
//
// Code generated for Simulink model 'levelController'.
//
// Model version                  : 1.9
// Simulink Coder version         : 8.13 (R2017b) 24-Jul-2017
// C/C++ source code generated on : Wed Oct 31 19:44:42 2018
//
// Target selection: ert.tlc
// Embedded hardware selection: Intel->x86-64 (Windows64)
// Code generation objectives:
//    1. Execution efficiency
//    2. RAM efficiency
// Validation result: Not run
//
#include "controlLevel.h"

// Include model header file for global data
#include "levelController.h"
#include "levelController_private.h"

// Named constants for Chart: '<Root>/controlLevel'
#define IN_CLOSE                       ((uint8_T)1U)
#define IN_INITIALIZATION              ((uint8_T)2U)
#define IN_OPEN                        ((uint8_T)3U)

// Output and update for atomic system: '<Root>/controlLevel'
void levelControllerModelClass::controlLevel()
```

```
{
  // Chart: '<Root>/controlLevel' incorporates:
  //   Outport: '<Root>/y'

  if (rtDW.is_active_c3_levelController == 0U) {
    rtDW.is_active_c3_levelController = 1U;
    rtDW.is_c3_levelController = IN_INITIALIZATION;
    rtY.y.valveState = 0;
  } else {
    switch (rtDW.is_c3_levelController) {
     case IN_CLOSE:
      if (rtDW.BusConversion_InsertedFor_contr.level <
          rtDW.BusConversion_InsertedFor_contr.min) {
        rtDW.is_c3_levelController = IN_OPEN;
        rtY.y.valveState = 1;
      }
      break;

     case IN_INITIALIZATION:
      if (rtDW.BusConversion_InsertedFor_contr.level <
          rtDW.BusConversion_InsertedFor_contr.min) {
        rtDW.is_c3_levelController = IN_OPEN;
        rtY.y.valveState = 1;
      } else {
        if (rtDW.BusConversion_InsertedFor_contr.level >
            rtDW.BusConversion_InsertedFor_contr.max) {
          rtDW.is_c3_levelController = IN_CLOSE;
          rtY.y.valveState = 0;
        }
      }
      break;

     default:
      if (rtDW.BusConversion_InsertedFor_contr.level >
          rtDW.BusConversion_InsertedFor_contr.max) {
        rtDW.is_c3_levelController = IN_CLOSE;
        rtY.y.valveState = 0;
      }
      break;
    }
  }

  // End of Chart: '<Root>/controlLevel'
}
//
// File trailer for generated code.
//
// [EOF]
//
```

3. Integracja aplikacji regulatora

Finalny program tj. aplikacja regulująca poziom cieczy w zbiorniku zintegrowana została w środowisku Visual Studio. W tym celu stworzony został projekt, do którego podłączone zostały wygenerowane pliki źródłowe i nagłówkowe.

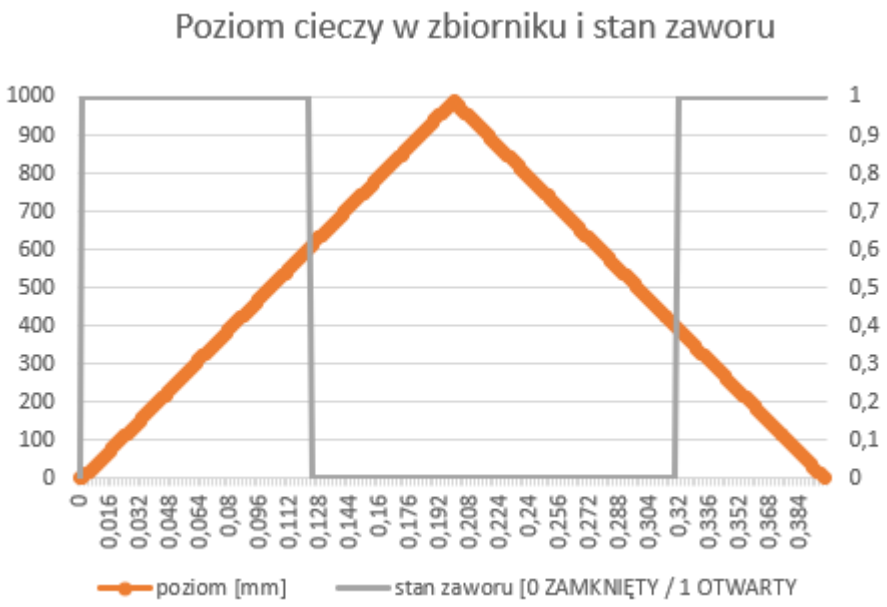
3.1. Funkcja main aplikacji

```
int main(int argc, char* argv[])
{

    Integer measuredLevel_mm = 0;
    Integer valvePosition    = 0;
    bool testResult          = true;

    Tank t;
    levelControllerModelClass regulator;

    t.setRequiredLevel(PRESET_LEVEL_MM, DELTA_LEVEL_MM);

    // input vector
    std::fstream fInput(argv[1], std::fstream::in);

    // regulation simulation
    while (!fInput.eof())
    {
           fInput >> measuredLevel_mm;
           fInput >> valvePosition;

           if (fInput.eof()) break;

           t.setCurrentLevel(measuredLevel_mm);

           regulator.rtU.u.level = t.currentLevel_mm;
           regulator.rtU.u.min   = t.levelMin_mm;
           regulator.rtU.u.max   = t.levelMax_mm;

           regulator.step();

           t.myValve[0].state = regulator.rtY.y.valveState;

           if (t.myValve[0].state != valvePosition)
           {
                  testResult = false;
           }
    }

    checkTestResult(testResult, argv[1]);
    fInput.close();
}
```

3.2. Testowanie jednostkowe zintegrowanego algorytmu

Funkcja main aplikacji zaprojektowana została w sposób pozwalający wywołać ją z linii komend wiersza poleceń lub poprzez plik wsadowy. Głównym zadaniem funkcji main jest doprowadzenie wejściowego wektora testowego do regulatora i sprawdzenie czy wynik działania programu jest dokładnie taki sam jak w wektorze wejściowym. Walidacja poprawności działania algorytmu sterującego przeprowadzona została poprzez wykonanie programu z parametrami wejściowymi przedstawionymi poniżej.

```
levelControl.exe testcase1.txt
levelControl.exe testcase2.txt
```



Rysunek 8. Wykres graficzny danych dla wektora przypadku testowego 2

W wyniku uruchomienia programu uzyskano dokładnie taki sam przebieg sygnału sterującego otwarciem i zamknięciem zaworu.

4. Podsumowanie i wnioski

Stosując generowanie kodu z modelów architektury i algorytmów sterujących zapewnia się dwukierunkowe śledzenie pomiędzy modelem a kodem co jest pewnego rodzaju dokumentacją kodu, która pozwala osobom bez znajomości języków programowania przeanalizować działanie programu. Kolejną zaletą tego podejścia jest redukcja zapotrzebowania na ręczne wytwarzanie kodu, który realizuje daną funkcjonalność. Niewątpliwym zyskiem w tym przypadku jest redukcja czasu

potrzebnego na wytworzenie i przetestowanie oprogramowania przed dostarczeniem go do odbiorcy. Zmniejszeniu ulega również ryzyko wystąpienia błędów funkcjonalnych związanych z niezgodnością kodu względem modelu.

Niestety generowanie kodu ma również swoje wady. Jedną z tych wad, w porównaniu do kodu wytworzonego manualnie i nierzadko zoptymalizowanego względem modelu, jest większe zapotrzebowanie na zasoby maszyny wykonującej daną aplikację; takich jak: pamięć swobodnego dostępu, pamięć nieulotna oraz czas procesora. Kolejną wadą kodu generowanego jest jego czytelność, która bardzo silnie zależy od ustawień danego programu generującego kod.

Dodatkowym wyzwaniem stawianym przed modelami oprogramowania przeznaczonymi do wytworzenia kodu generowanego jest spełnienie reguł wytwarzania bezpiecznego oprogramowania. W przypadku kodu wytwarzanego ręcznie jest to MISRA[4]. W przypadku modeli przeznaczonych go generowania kodu są to reguły MISRA AGC[5] i MAAB[6].

LITERATURA

1. Serwis internetowy Automotive SPICE – Process Reference Model v4.5 : *http://www.automotivespice.com/fileadmin/software-download/automotiveSIG_PRM_v45.pdf,* 23.10.2018
2. Serwis internetowy Object Management Group – UML 1.4 : *https://www.omg.org/spec/UML/1.4/*, 31.10.2018
3. Serwis internetowy ArgoUML – Documentation : *http://argouml.tigris.org/documentation/,* 31.10.2018
4. Serwis internetowy Mathworks – Embedded Code Generation : *https://www.mathworks.com/solutions/embedded-code-generation.html,* 31.10.2018

[4] Motor Industry Software Reliability Association
[5] Motor Industry Software Reliability Association Auto Generated Code
[6] Mathworks Automotive Advisory Board