European Coordination for Accelerator Research and Development

PUBLICATION

Object oriented programming environment for reconfigurable applications implemented in FPGA chips

Drabik, P (Warsaw U. of Tech.) *et al*

02 September 2012

Elektronika

The research leading to these results has received funding from the European Commission under the FP7 Research Infrastructures project EuCARD, grant agreement no. 227579.

This work is part of EuCARD Work Package **4: AccNet: Accelerator Science Networks**.

The electronic version of this EuCARD Publication is available via the EuCARD web site <http://cern.ch/eucard> or on the CERN Document Server at the following URL : <http://cdsweb.cern.ch/record/1235145

Object oriented programming environment for reconfigurable applications implemented in FPGA chips
Obiektowe środowisko programowania wspierające systemy rekonfigurowalne implementowane w układach FPGA

PREPRINT Elektronika 50, 8 (2009) pp. 183-187

P.Drabik, K.Poźniak, ISE WUT

Wielokanałowe, rozproszone systemy elektroniczne coraz częściej są konstruowane na bazie układów typu *Field Programmable Gateway Arrays* (FPGA). Układy FPGA zawierają dużą liczbę uniwersalnych bloków logicznych, modułów pamięci, bloków DSP, szybkich interfejsów komunikacyjnych itp. Mogą być wielokrotnie rekonfigurowane. Uzyskano w ten sposób możliwość modyfikacji części funkcjonalnej systemu pomimo jego niezmiennej struktury sprzętowej i umiejscowienia na obiekcie.

W układach FPGA są coraz częściej implementowane parametryzowane, biblioteczne moduły funkcjonalne. Dzięki temu rozwiązaniu można adaptować te same moduły do zmienionych uwarunkowań, bądź zupełnie nowych zastosowań. W celu efektywnego wprowadzania takich zmian, opracowano sparametryzowany opis struktury sprzętowej w postaci *pliku definicji*. Zawartość tego pliku odwzorowuje moduły sprzętowe poprzez listę ich parametrów i interfejsów. Na podstawie pliku definicji generowany jest odpowiedni plik konfiguracyjny, np. w języku VHDL.

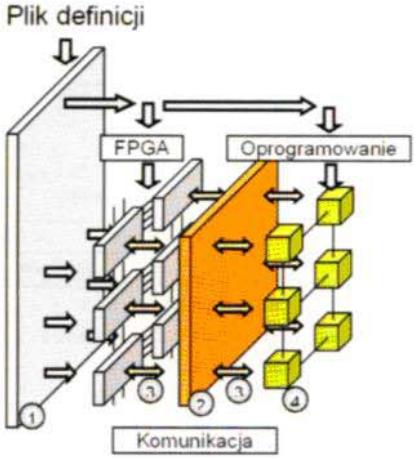
Uzyskana w ten sposób struktura sprzętowa wymaga poprawnego sterowania z poziomu aplikacji komputerowych. Na bazie *pliku definicji* są generowane struktury w warstwie programistycznej, relewantne do struktur sprzętowych. Schemat na *rys. 1* w ogólny sposób pokazuje zasadę generowania struktury systemu z pliku definicji. Plik ten określa moduły sprzętowe (1) w systemie, a tym samym zostają określone ich klasy w oprogramowaniu (4). Poprzez interfejs komunikacyjny (2), obiekty tych klas mogą się bezpośrednio komunikować (3) z ich sprzętowymi odpowiednikami. W ten sposób zostało zapewnione sterowanie poszczególnymi modułami sprzętowymi poprzez obsługę ich obiektowych odpowiedników w oprogramowaniu.

Interfejs komunikacyjny Component Internal Interface

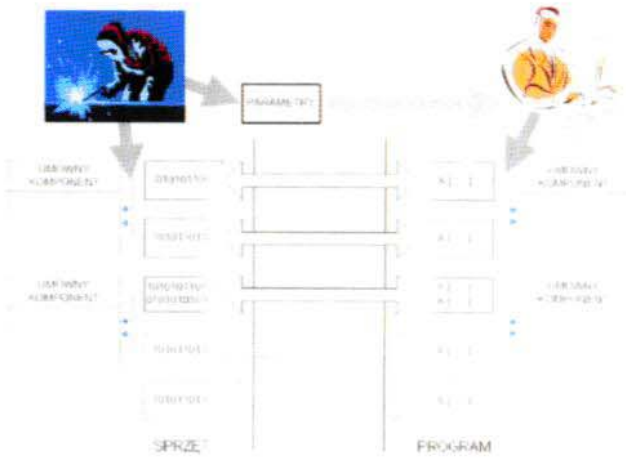
Metoda nazwana *Component Internal Interface* (CII) jest jednym z przykładów zastosowania sparametryzowanej definicji sprzętu i oprogramowania. Została opracowana w celu zapewnienia efektywnej integracji warstwy sprzętowej, zaimplementowanej w układach FPGA, z warstwą oprogramowania komputerowego (np. aplikacji pracującej na komputerze PC). Koncepcja CII polega na opracowaniu tożsamych struktur, które wiążą w jednoznaczny sposób definicje sprzętową i programową. CII jest funkcjonalnym rozwinięciem technicznego interfejsu komunikacyjnego nazwanego *Internal Interface* (II). Dlatego bliższe omówienie CII poprzedzono krótkim wprowadzeniem do budowy interfejsu II. Na *rys. 2* przedstawiono ogólną zasadę jego działania [1,2].

Podstawowym mechanizmem II jest komunikacja modułów sprzętowych z ich odpowiednikami w oprogramowaniu. Procesy te przedstawione w formie strzałek łączących ze sobą rejestry, bloki pamięci (po lewej stronie schematu) oraz obiekty i funkcje oprogramowania (po prawej stronie schematu). Integracja funkcjonalności polega na arbitralnym połączeniu przez projektantów odpowiednich zasobów (zarówno od strony sprzętu, jak i oprogramowania). Ponadto, w konfiguracji II parametry są globalne i nie powiązane z formalnymi komponentami (obiektami). Parametry powiązane z określonym modułem są arbitralnie definiowane i muszą być stosowane umownie w warstwie programistycznej. W konsekwencji trzeba się posługiwać umownymi komponentami (obiektami).

Implementacja systemów w układach FPGA sprowadza się do wyspecyfikowania w pliku definicji umownych komponentów umieszczonych w sprzęcie poprzez listę ich interfejsów i umowny zestaw parametrów. Na podstawie pliku



Rys. 1. Blokowy schemat systemu
Fig. 1. System overview



Rys. 2. II - koncepcja realizacji Fig. 2. II - conception overview

definicji następuje generacja kodu źródłowego, zawierającego specyfikację komunikacji dla sprzętu (w języku VHDL) oraz dla oprogramowania (języki: C++, Java, lub Matlab). W specyfikacji są zdefiniowane interfejsy (np. adres dostępu, szerokość magistrali itp.) oraz parametry globalne. Na tej podstawie są automatycznie integrowane struktury sprzętowe powiązane ze strukturami oprogramowania (co obrazują poziome strzałki na *rys. 2*).

II jest rozwiązaniem prostym funkcjonalnie, dedykowanym przede wszystkim dla statycznej konfiguracji portów I/O. W przypadku, gdy zachodzi potrzeba dodania, bądź odłączenia określonych modułów funkcjonalnych, wiąże się to z koniecznością modyfikacji plików źródłowych dla sprzętu i oprogramowania. Jeśli modyfikowana funkcjonalność posiada wiele referencji obiektu umownego z zasobami sprzętowymi, zachodzi potrzeba modyfikacji kodu we wszystkich miejscach jego użycia, zarówno w warstwie sprzętu, jak i dla oprogramowania. To jest nie tylko czasochłonna operacja, ale i bardzo prawdopodobne źródło wprowadzania błędów.

Jedną z podstawowych niedogodności II jest liniowy sposób definiowania modułów poprzez komponenty umowne. Powoduje to w konsekwencji brak elastyczności i automatyczności przystosowania warstwy sprzętowej i programistycznej do częstych zmian w strukturach systemu złożonego z wielu modułów połączonych w konfiguracji drzewiastej.

Bardziej rozwiniętą realizacją sparametryzowanej definicji sprzętu oraz oprogramowania jest wspomniany interfejs CII. Realizacja tego interfejsu bazuje na mechanizmie II (przedstawionym na *rys. 2*) i jest wzbogacona o parametryzację definicji wydzielonych elementów systemu. Dzięki takiemu podejściu powstają definicje komponentów, które wraz z ich wzajemnymi relacjami kreują cały system pomiarowy, jak to zobrazowano na *rys. 3*.

Utworzone w projekcie komponenty są sparametryzowane i w naturalny sposób tworzą strukturę drzewa. Mogą one posiadać „ojca" - komponent nadrzędny i „dzieci" - będące komponentami zależnymi (nazywane także subkomponentami). W ten sposób ich jednokrotna definicja może być wiele razy użyta (instancjowana) dla różnych parametrów wywołania. Dzięki powyższemu podejściu, przykładowe dodanie nowego interfejsu do jednego z modułów sprowadzi się w praktyce do dodania treści jedynie w definicji komponentu. Zmiany zostaną uwzględnione automatycznie, w każdym miejscu, gdzie ten komponent został użyty (instancjowany).

Ważką cechą mechanizmu CII jest zawarcie parametru krotności występowania komponentu. Oznacza to, że definiując krotność równą 0, fizycznie usuwa się komponent z systemu, jednak jego formalna definicja pozostaje. Ta cecha może pozwolić na szybką i efektywną rekonfigurację logiki systemu (włączanie lub wyłączanie określonych modułów wykonawczych). Kolejną cechą CII jest uzyskanie komponentowej, obiektowo zorientowanej architektury. Wynika z tego przejrzysty obraz funkcjonalności i możliwości implementacji systemu, a także procesu wytwórczego (obraz zależności i wzajemnych relacji w pliku konfiguracyjnym). Mechanizm CII udostępnia spójną komunikację aplikacji komputerowej z interfejsem sprzętowym na poziomie architektury.

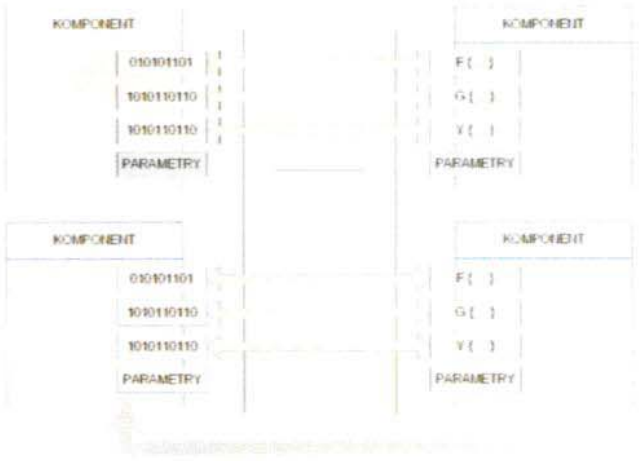
Oprogramowanie interfejsu CII

Układy FPGA umożliwiają wielokrotną i szybką rekonfigurację warstwy sprzętowej. Podobnych właściwości oczekuje się od warstwy programistycznej. Do pełnej konstrukcji systemu zgodnego z CII opracowano warstwę oprogramowania w języku C++. Ważką cechą takiego oprogramowania jest jego *generyczność*. Paradygmat programowania generycznego [3] umożliwia projektowanie rozwiązań, podstawą których jest architektura, a nie o jednoznacznie zdefiniowane struktury danych. Uzyskuje się możliwość budowania aplikacji bez skonkretyzowanej, lecz jedynie ogólnej wiedzy na temat jej struktur danych. W przypadku architektur systemów elektronicznych są to różne typy komponentów - odpowiedniki modułów sprzętowych. Próba określenia struktur danych stanowiłaby ograniczenie dla całej koncepcji systemu. Zastosowanie *polimorfizmu* [3] oraz specjalnych wzorców programowania [4] umożliwia programistom dokonywanie zmian funkcjonalności obiektów komponentów. Wykorzystanie naturalnej obiektowości tego języka pozwala na tworzenie hierarchii klas i interfejsów, która ułatwia poprawne projektowanie programów oraz wielokrotne użycie klas.

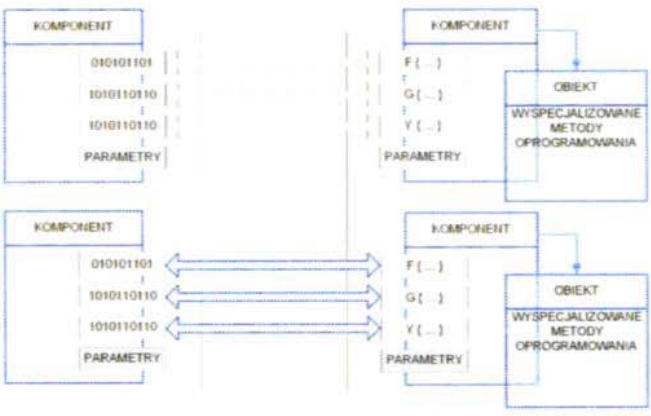
Na *rysunku 4*. przedstawiono metodę dostępu do sprzętu poprzez interfejs CII z poziomu oprogramowania. W projekcie zapewniono możliwość dodawania własnych wyspecjalizowanych funkcji. Są one dostępne w klasach dziedziczących z klas komponentów. Takie podejście gwarantuje pracę na prawidłowych obiektach odwzorowanych w sprzęcie.

Zadaniami projektu dla aplikacji administrujących systemy, bazujące na interfejsie CII są:

- dostarczenie efektywnego mechanizmu do administracji systemem implementowanym w układzie FPGA,



Rys. 3. CII - koncepcja zagadnienia
Fig. 3. CII - conception overview



Rys. 4. Aplikacja użytkowa w C++
Fig. 4. Application conception for C++

- udostępnienie zestawu bibliotek i plików nagłówkowych, umożliwiających pełne wykorzystanie mechanizmów CII,
- udostępnienie zestawu bibliotek i plików nagłówkowych bazujących na wygenerowanych plikach CII, pozwalających programistom rozwinąć je o własne funkcje i zastosowania.

Architektura systemu pomiarowego bazuje na komponentach oraz ich wzajemnych relacjach. Komponenty tworzą strukturę drzewa. Mechanizm określenia miejsca w tym drzewie został zaimplementowany w klasach obiektów. Utworzenie hierarchii obiektów (stanowiących obraz komponentów w układach FPGA) poprzez mechanizm dziedziczenia umożliwia budowanie własnych implementacji komponentów.

Dotychczas, w procesie kreowania systemu, generowane były pliki definicji sprzętu i oprogramowania. Podczas omawianego procesu, generator dostarczał ściśle określone implementacje obiektów implementowanych w FPGA. Były to obiekty przystosowane jedynie do komunikacji ze sprzętem. Nie zawierały żadnej zaawansowanej funkcjonalności, potrzebnej użytkownikom tego typu systemów.

Udostępnienie funkcjonalności hierarchii klas pozwoliło na znaczne zautomatyzowanie procesu modyfikacji systemu. Ponadto, pozwoliło rozbudowywać system o operacje wyższego poziomu. Tworząc własne implementacje komponentów, programiści mogą łatwo dodać różnego rodzaju operacje. Nie jest wymagane posiadanie szczegółowej wiedzy o sprzętowej strukturze tzw. *komponentu standardowego*, dla którego realizują implementacje. Przykładowo, można utworzyć metodę w nowym komponencie, ale wynikającym z uprzedniego (dzięki hierarchii klas) i używać jej w miejscach występowania tego komponentu. Efektywnym wzorcem programowania, udostępniającym taką funkcjonalność (tzn. tworzenia obiektów, których typy byłyby określane dynamicznie) jest wzorzec *AbstractFactory* [4].

Oprogramowanie CII zorientowane obiektowo - implementacja wzorca AbstractFactory

Zasadą zachowania tego wzorca programowania jest udostępnienie metody do fabrykowania nowych obiektów ze specjalnej klasy *factory*. W przedstawianym projekcie, wzorzec ten został wzbogacony o strukturę rejestrującą nowe typy obiektów, jakie będą tworzone przez *factory*. Istnieje możliwość użycia wygenerowanych typów komponentów, a także mechanizmu dynamicznego rejestrowania nowych typów, które będą produktami *factory*. Umożliwia to dedykowana w tym celu struktura, której przykładową implementację przedstawiono na *rys. 5*.

Tego typu struktura umożliwia utworzenie wspólnego rejestru wszystkich *factory*, definiując w nim typ klasy oraz jej identyfikator. Jest on typu definiowanego, który w tym przypadku stanowi łańcuch znaków.

typedef std::string TypeIdType;

```
template <class T>
        class RegisterFactory
        {
        public:
                RegisterFactory( FactoryRegister & factoryRegister
                                , TypeIdType typeId
                                )
                {
                        BaseComponentFactory * pFactory = new ComponentFactory <T>;
                        factoryRegister.RegisterFactory( typeId, pFactory );
                }
        };
```

Rys. 5. Przykład kodu źródłowego szablonu klasy *RegisterFactory* **Fig. 5. *RegisterFactory* class listing (part)**

```
class BaseComponentFactory {
public:
    virtual ~BaseComponentFactory() {
    }

    virtual Component * MakeObject(const CCII_CONFIG_TABLE *tab_cfg,
            long base_width, int pos, tb_cii::AccessHardwareConfig& chn) = 0;
};

template<class FactoryType> class ComponentFactory : public BaseComponentFactory {
public:
    virtual Component * MakeObject(const CCII_CONFIG_TABLE *tab_cfg,
            long base_width, int pos, tb_cii::AccessHardwareConfig& chn) {
        return new FactoryType(tab_cfg,base_width,pos,ComponentInfo::getInstancePointer(),chn);
    }
};
```

Rys. 6. Listing kodu źródłowego klasy *BaseComponentFactory* oraz szablonu *ComponentFactory*
Fig. 6. *BaseComponentFactory* class and *ComponentFactory* template listing (part)

```
class FactoryRegister {
public:
    virtual ~FactoryRegister() {
        std::for_each(Factories_.begin() , Factories_.end() , DeleteFactory);
    }

    Component * NewObject(TypeIdType typeId, const CCII_CONFIG_TABLE *tab_cfg,
            long base_width, int table_pos, tb_cii::AccessHardwareConfig& chan) {
        Component * pObject = NULL;

        FactoryMapType::iterator pos = Factories_.find(typeId);

        if (pos != Factories_.end() && pos->second != NULL) {
            pObject = pos->second->MakeObject(tab_cfg, base_width, table_pos,
                    chan);
        }

        return pObject;
    }

    void RegisterFactory(TypeIdType typeId, BaseComponentFactory * factory) {
        Factories_.insert(std::make_pair(typeId, factory) );
    }

private:
    typedef std::map<TypeIdType, BaseComponentFactory*> FactoryMapType;

    FactoryMapType Factories_;
};
```

Rys. 7. Listing kodu źródłowego klasy *FactoryRegister* **Fig. 7. *FactoryRegister* class listing (part)**

Zasadą działania *RegisterFactory* jest utworzenie klasy *factory* dla danego typu, określonego przez programistę w obiekcie *FactoryRegister* (linia 69 na *rys. 5*). Jest to programistyczne udogodnienie, dostosowane do użytego wzorca. Zasada zachowania wzorca sprowadza się do tworzenia klas *factory* dla każdego typu, który ma być przez tę klasę udostępniony. Stosując to udogodnienie należy jedynie zarejestrować odpowiednią klasę w rejestrze *RegisterFactory*. W przypadku CII odnosi się to do klas dziedziczących z *komponentu standardowego*. Obiekt *factory* jest wtedy automatycznie tworzony.

Szablon klas *RegisterFactory* używa obiektów typu *BaseComponentFactory* oraz *ComponentFactory*. Definicję tych klas przedstawiono na *rys. 6*.

Rysunek 6. przedstawia dalszą część implementacji wzorca *AbstractFactory* w postaci klasy o nazwie *BaseComponentFactory*, która posiada wirtualną metodę do tworzenia oraz zwracania obiektów danej *factory* (linia 251). Dla określonych zastosowań należy tę klasę zrealizować poprzez dziedziczenie i implementacje metod wirtualnych. W ten sposób działa szablon klas *ComponentFactory*. Został on przystosowany w taki sposób, aby tworzyć *factory* dla każdego z typu implementującego hierarchię. Przystosowanie to uwidacznia się w konstruktorze obiektów zwracanych przez *factory* (linia 259). Jest on identyczny dla każdego z typów komponentów CII, ponieważ tylko wówczas można w prosty sposób wykorzystać omawiany wzorzec.

Implementację wzorca realizuje klasa *FactoryRegister* (*rys. 7*). Została w niej zakodowana metoda *RegisterFactory(...)*, którą wykorzystuje rejestr *factory*. Jak to pokazano w linii 21, operacja polega na umieszczeniu obiektu *factory* w specjalnej mapie (*FactoryMapType*, linia 26) według identyfikatora. W celu pozyskania obiektu oczekiwanej klasy, programista ma udostępnioną metodę *NewObject(...)*. Szuka ona wzorca *factory* względem podanego typu (linia 11), a następne tworzy obiekt zdefiniowanej klasy. Uzyskany obiekt może być używany standardowo lub w sposób zaawansowany. W takym ostatnim przypadku, poprzez instrukcję `dynamic_cast<>` należy wskazać typu obiektu.

Za pośrednictwem dotychczas omówionych mechanizmów użytkownik może w dogodny sposób kreować wymagane funkcje, bazując na wygenerowanym kodzie. W ten sposób otrzymuje dostęp do właściwych portów I/O w układach FPGA z poziomu aplikacji w C++ oraz zawartych w niej obiektów i metod. Przykładowo, operatorzy mogą bezpośrednio zapisywać dane do rejestrów (zmieniając np. parametry pracy układu) lub monitorować rezultaty działania układu (np. poprzez odczyt rejestrów lub pamięci).

Pilotażowa implementacja CII dla Trygera Mionowego RPC

Opracowane pliki nagłówkowe w języku C++ oraz mechanizmy do obsługi interfejsu sprzętowego CII, pozwoliły opracować programy do testowania funkcji modułów dla systemu Trygera Mionowego RPC. Tryger Mionowy RPC [5,6] stanowi integralną część eksperymentu CMS [7] przy akceleratorze LHC [8] (CERN). Badania są poświęcone poznaniu struktury materii, a w szczególności wykryciu cząstki Higgsa. Tryger Mionowy RPC obsługuje prawie 200 000 mionowych kanałów detekcyjnych, przy częstości 40 mln. pomiarów na każdy kanał niezależnie. Jego zadaniem jest szybki pomiar pędu poprzecznego wytworzonych mionów i wyszukiwaniu czterech mionów o najwyższych energiach. System Trygera Mionowego RPC jest reprezentatywnym przykładem rozległych, systemów pomiarowych. Składa się z kilku tysięcy oddzielnych modułów elektronicznych, na których osadzono ponad 10 000 układów FPGA firmy Altera i Xilinx.

Wykorzystując mechanizmy opisane wcześniej, wykonano aplikację testującą działanie CII. Do testów wykorzystano część elektroniki Trygera Mionowego RPC odbierającą strumienie danych pomiarowych z łączy optycznych. W aplikacji utworzono obiekty implementujące zaawansowane funkcje do przeprowadzania operacji komunikacji z modułami sprzętowymi. Zbadano działanie *komponentów standardowych* oraz komponentów rozszerzonych funkcjonalnie. Przeprowadzone testy wykazały poprawność komunikacji oraz pełną integrację sprzętu z oprogramowaniem. Potwierdziły przydatność zastosowania wytworzonej biblioteki we współpracy z systemem zbudowanym na bazie układów FPGA.

Dalszy rozwój przedstawionej metodyki umożliwi użytkownikom oprogramowania rozbudowanie funkcjonalności bezpośrednio wbudowywanych w warstwę interfejsu sprzętowego. Zwiększona zostanie efektywna kontrola rozproszonego systemu pomiarowego, zrealizowanego na bazie wielu układów FPGA.

Podsumowanie

W artykule przedstawiono autorski przykład rozwiązania zagadnienia sterowania złożonymi i rozproszonymi systemami elektronicznymi bazującymi na układach FPGA. Wymagana jest nie tylko poprawna konfiguracja systemu, ale także dokładna jego synchronizacja, bieżący monitoring oraz diagnostyka. Dyskutowane funkcjonalności zapewniają zarówno oprogramowanie warstwy sprzętowej zaimplementowanej w układach FPGA (*firmware*), jak i warstwy oprogramowania współpracującego z systemem. Omówiono metodykę integracji modułów sprzętowych FPGA z oprogramowaniem, którego zaletą jest dynamika dodawania funkcjonalności zarządzających elektroniką zgromadzonych w bibliotekach użytkownika. Pilotażowa implementacja biblioteki potwierdziła poprawność koncepcji oraz możliwości rozwojowe opracowanej aplikacji.

Literatura

[1] Pozniak K. T., Bartoszek M., Pietrusinski M.: Internal interface for RPC muon trigger electronic at CMS experiment. Proc. SPIE 5484, pp. 269-82, 2004.
[2] Pozniak K. T.: "Internal Interface", TESLA Report 2005-22, 2005.
[3] Kayshav Dattatri. Język C++. Efektywne programowanie obiektowe. Helion 2005.
[4] James William Cooper. Java. Wzorce projektowe. Helion 2001.
[5] CMS MUON Technical Design Report, CERN/LHCC 97-32, CMS TDR 3, 1997.
[6] Kalinowski A., Krolikowski J., Zych P.: Muon Trigger Algorithms Based on 6 RPC Planes. CMS NOTE-2001/045, CERN, 2001.
[7] http://cmsinfo.cern.ch/Welcome.html - [CMS Experiment Home Page]
[8] http://www.lhc01.cern.ch/ - [LHC homepage]