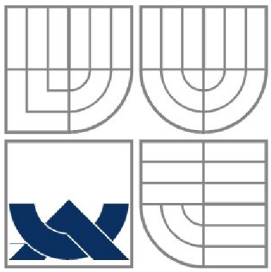
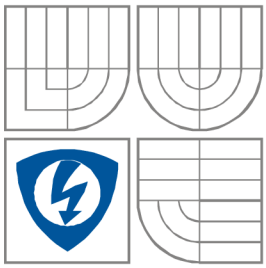


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

ROZŠÍŘENÍ NORMY IEC 61131-3 O OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ PLC

OBJECT ORIENTED PLC PROGRAMMING EXTENSION OF IEC 61131-3 STANDARD

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

Ing. DANIEL GROMUS

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN PÁSEK, Csc.

BRNO 2011

Originální zadání diplomové / bakalářské práce

Abstrakt

Úvodní část tato práce obsahuje stručný popis normy IEC 61131-3 a jejich navrhovaných rozšíření o prvky objektově orientovaného programování. V následující části je zhodnocení možnosti použití konceptu událostně řízeného programování. V poslední části je ukázáno několik praktických příkladů využití prvků objektově orientovaného programování.

Klíčová slova

IEC 61131-3, objektově orientované programování, událostně řízené programování, CoDeSys

Abstract

The first part of this thesis contains brief description of the IEC 61131-3 standard and proposed extension of object oriented programming to it. There is a review of possibility using event driven programming concept in the next part. The last part shows some practical examples taking use of object oriented programming elements.

Keywords

IEC 61131-3, object oriented programming, event driven programming, CoDeSys

Bibliografická citace:

GROMUS, D. *Rozšíření normy IEC 61131-3 o objektově orientované programování PLC*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2011. 47s. Vedoucí bakalářské práce byl Ing. Jan Pásek, CSc.

Prohlášení

„Prohlašuji, že svou bakalářskou práci na téma rozšíření normy IEC 61131-3 o objektově orientované programování jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této bakalářské práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

V Brně dne: **23. května 2011** .....

podpis autora

Obsah

Seznam obrázků

1 ÚVOD

V současné době jsou kladeny větší a větší požadavky na řídící systémy. Řídící systémy musí plnit stále více a více různých funkcí. Stále častěji se hledají univerzální, plně konfigurovatelná a opakovatelná řešení. Výsledkem toho jsou velké, komplexní průmyslové aplikace. Na druhé straně je však zkracován čas na jejich tvorbu a nasazení. To nutí jejich tvůrce k zavádění postupů vedoucí k aplikacím složených z obecných znovupoužitelných částí kódu, které lze rychle začlenit do nově vytvářených aplikací.

Velká část dnešních řídících automatů se programuje v jazycích standardizovaných normou IEC 61131-3. Ta sice obsahuje způsob strukturování kódu, ale ten je velmi omezený. Pokud se však podíváme do světa informačních technologií, zjistíme, že již existuje propracovaný a ověřený koncept označovaný jako objektově orientované programování. Ten používá většina současných vyšších programovacích jazyků jako je např. C++ nebo Java. A právě rozšíření o prvky objektově orientovaného programování se objevuje i v návrhu na novou třetí verzi normy IEC 61131-3.

Cílem této práce je popis připravovaného rozšíření normy. Dále pak srovnání současného přístupu k programování řídících automatů a přístupu s možností využití objektově orientovaného programování.

2 OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ

V této kapitole bych chtěl stručně popsat myšlenku objektově orientovaného programování a pojmů s tím souvisejících. Blíže viz [1] [2] [10].

Objektově orientované programování je způsob programování, při kterém se dělení a strukturování výsledného kódu inspiruje modelem reálného světa. V reálném světě jsou věci, které mají své vlastnosti a plní nějaké činnosti. Obrazem takové věci v programu je s jistou mírou abstrakce objekt. Objekt reprezentuje konkrétní věc, obsahuje její funkčnost zná její vlastnosti a pamatuje si její vnitřní stav. Jednotlivé objekty lze vzájemně skládat dohromady a vytvářet tak další komplexnější objekty. Výsledný program se potom skládá ze skupiny vzájemně provázaných a spolupracujících objektů.

Důvodem pro použití objektově orientovaného programování je především snaha získat modulární, robustní a snadno udržovatelný kód. Při správném návrhu je možná alespoň částečná znovupoužitelnost dříve vytvořeného kódu. V tomto smyslu se na vytvořené objekty nahlíží jako na základní stavební prvky, ze kterých lze budovat různě složité programy určené k často odlišným účelům.

2.1 Objekt

Objekt se skládá ze dvou částí. První část jsou data, která slouží jako paměť vnitřního stavu objektu a která jsou přístupná pouze objektu samotnému, popřípadě je vystavuje vně pomocí vlastností. Druhá část je potom funkčnost, kterou objekt poskytuje navenek pomocí metod. Předpisem objektu je jeho třída. Třída obsahuje definici jak jednotlivých datových položek, tak i metod. Na třídu lze nahlížet jako na šablonu objektu. Za běhu programu jsou podle tříd vytvářeny jednotlivé instance objektů. Pro každou instanci je v paměti vyčleněna část pro uložení vlastních datových položek, tak jak bylo definováno v její třídě. Kód metod dané třídy je pro všechny její instance stejný, ale vždy pracuje s daty té konkrétní instance objektu, pro kterou byla metoda volána.

2.2 Zapouzdření

Význam zapouzdření je ve skrytí vnitřní implementace objektu a umožnění přístupu k objektu pouze přes definované vnější rozhraní. Objekt se tak z vnějšku tváří jako černá skříňka, která nabízí definované metody a vlastnosti. Jednotlivé metody a vlastnosti objektu lze vyvolat, ale v žádném případě není z vnějšku možné ovlivnit jakým způsobem jsou uvnitř objektu vykonány.

2.3 Vazby mezi objekty

Jak již bylo zmíněno, mezi objekty se v programu vytvářejí vazby. Vazbou je s ohledem na zapouzdření myšleno volání metod jiného objektu. Tato vazba může být horizontální, kdy se objekty nacházejí vedle sebe a vzájemně používají své metody. Nebo může být vazba vertikální, kdy jeden objekt je součástí jiného objektu. Potom pouze nadřazený objekt může volat metody vnořeného objektu.
Dalšími typy vazeb jsou použití rozhraní a dědičnost.

2.4 Rozhraní

Myšlenka vazeb mezi objekty přes definované rozhraní je ta, že objekt nevytváří přímou vazbu s jiným konkrétním objektem, ale pouze požaduje, aby ten objekt měl danou funkčnost. To znamená, že takto definovanou vazbu může vytvořit z různými objekty, ale každý z nich musí mít implementované požadované rozhraní. Požadované rozhraní musí být v programu definováno. Jedná se ale pouze o deklaraci rozhraní, která neobsahuje žádnou implementaci. Implementace je až v objektech, kteří přijmou dané rozhraní. V souvislosti s přijetím rozhraní se někdy mluví jako o dědičnosti rozhraní.

2.5 Dědičnost

Dědičnost je vazba mezi objekty, kdy jeden objekt rozšiřuje funkčnost jiného objektu. Potom se mluví o původním objektu jako o rodiči (nebo základní třídě) a o rozšiřujícím objektu jako o potomku (nebo odvozené třídě). Potomek dědí a automaticky plní veškerou funkčnost jako jeho rodič navíc tuto funkčnost může změnit nebo rozšířit. To znamená, že potomek má stejné vlastnosti a metody jako rodič, ale může překrýt jejich implementaci nebo přidat zcela nové. Častým scénářem je, kdy rodič je ve vztahu k potomkům obecný objekt a jednotliví její potomci jsou již jeho speciální odvozeniny. Význam dědičnosti spočívá především v možnosti znovupoužitelnosti kódu rodiče.

2.6 Polymorfizmus

Polymorfizmus souvisí s dědičností a vyjadřuje, že všude tam, kde je ve vazbě použit rodičovský objekt, lze místo něj použít jeho potomka. Jak již bylo uvedeno potomek plní veškerou funkčnost svého rodiče, ale může ji překrýt svou vlastní implementací. V tom případě bude vykonána implementace potomka.

S využitím polymorfizmu lze tedy vytvářet volnější vazby vedoucí k obecnějšímu návrhu.

3 NORMA IEC 61131-3

Předmětem normy IEC 61131 jsou programovatelné řídící systémy. V Evropské unii je tato norma přijata pod označením EN IEC 61131 v České Republice jako ČSN EN 61131. První verze normy byla vydána v roce 1993 a její druhá verze v roce 2003. V současnosti se pracuje na její třetí verzi, kde by mimo jiného měla být zapracována i podpora objektově orientovaného programování.

Norma se skládá z několika částí:

1. Všeobecné informace
2. Požadavky na zařízení a zkoušky
3. Programovací jazyky
4. Podpora uživatelů
5. Komunikace
6. Komunikace prostřednictvím sběrnic
7. Programování fuzzy řízení
8. Technické zprávy

Programovacími jazyky se zabývá třetí část normy [3] označována jako IEC 61131-3. Obsahuje specifikaci syntaxe a sémantiky ucelené skupiny programovacích jazyků pro programovatelné řídící automaty. Jejím cílem je standardizovat programovací jazyky řídících systémů. V současné době ji v různém rozsahu přijala většina větších výrobců PLC. Hlavním přínosem IEC 61131-3 je možnost programovat různé PLC různých výrobců stejnými programovacími jazyky a přenositelnost kódu mezi nimi.
Norma IEC 61131-3 ve své první části definuje základní prvky programovacího modelu a ve druhé části pak čtyři různé programovací jazyky. Blíže viz [5].

3.1 Základní prvky

Norma definuje základní datové typy. Podobně jako v jiných jazycích se jedná o binární, číselné, časové nebo znakové datové typy. Z nich lze dále vytvářet uživatelské odvozené typy jako jsou například struktury.

Jednotlivým datovým objektům v paměti lze připojit jména pomocí proměnných. Proměnné mohou být jedno prvkové nebo více prvkové (například typu pole nebo struktury). Proměnné lze vytvářet v rámci globální paměti nebo v deklarační části programových organizačních jednotek.

Jako programovou organizační jednotkou (POU) definuje norma funkci, funkční blok nebo program. Jak již bylo zmíněno POU obsahuje deklarační část. Druhou částí POU je pak programový kód. Proměnné tvořené v deklarační části lze rozdělit do skupin podle jejich použití a to na:

- vnitřní proměnné `VAR`, které jsou přístupné pouze uvnitř POU
- vstupní proměnné `VAR_INPUT`, přes které jsou předávány hodnoty z vnějšku do

POU

- výstupní proměnné `VAR_OUTPUT`, přes které jsou předávány hodnoty z POU na vnějšek
- vstupně/výstupní proměnné `VAR_IN_OUT`, které slouží zároveň jako vstupní i výstupní proměnné
- dočasné proměnné `VAR_TEMP`

Funkce má definovanou jednu návratovou proměnnou a jednu nebo více vstupní proměnných. Funkce slouží ke zpracování hodnot vstupních proměnných a vrácení výsledku na zásobník. Funkce je bezstavová. To znamená, že pokud má funkce ke své činnosti deklarované nějaké vnitřní proměnné, budou jejich hodnoty resetovány před každým voláním funkce. Tím je zaručeno, že výsledek volání funkce nebude ovlivněno jejími předchozími voláními a stejná kombinace hodnot vstupních proměnných vede vždy na stejný výsledek.

Funkční blok na rozdíl od funkce má vnitřní stav. Dále nemá definovanou návratovou hodnotu, ale může mít definovaný libovolný počet výstupních proměnných. Vnitřní stav tvoří vnitřní, výstupní a v závislosti na implementaci i vstupní proměnné. Funkční blok je instanční, podobně jako u tříd jsou vytvářeny různě pojmenované instance jednoho funkčního bloku, kdy každá instance má svůj vlastní vnitřní stav.
Pokud máme deklarované potřebné proměnné, funkce a instance funkčních bloku, vše spojíme a logiku řízení implementujeme do programu. Programy podobně jako funkční bloky mohou mít deklarované vstupní, výstupní a vnitřní proměnné. Instance programu mohou být vytvářeny pouze v souvislosti s přiřazením k některému z procesů nebo úloh.

Konfigurační prvky slouží především k nastavení využití výpočetních jednotek a paměti řídícího systému. Výpočetní výkon lze většinou konfigurovat na přepínání mezi zpracováním různých procesů a úloh, kterým lze přiřadit konkrétní programy. Taktéž většina systémů má část paměti uživatelsky přístupnou pro deklaraci globálních proměnných používaných v programech. Možnosti konfigurace jsou přímo závislé na konkrétním řídícím systému a mohou se i výrazně lišit.

3.2 Programovací jazyky

Norma definuje hned několik různých programovacích jazyků pro řídící systémy. Dva textové

- Instrukce (Instruction List - IL)
- Strukturovaný text (Structured Text - ST)

a dva grafické

- Reléová schémata (Ladder Diagram - LD)
- Funkční bloky (Function Block Diagram - FBD)

Dále definuje nástroj pro vytváření sekvenčního řízení (Sequential Function Chart -

SFC), který slouží k definování posloupnosti s jakou se mají jednotlivé části programu vykonávat. Implementace těchto částí programu pak může být v libovolném ze zmíněných jazyků.

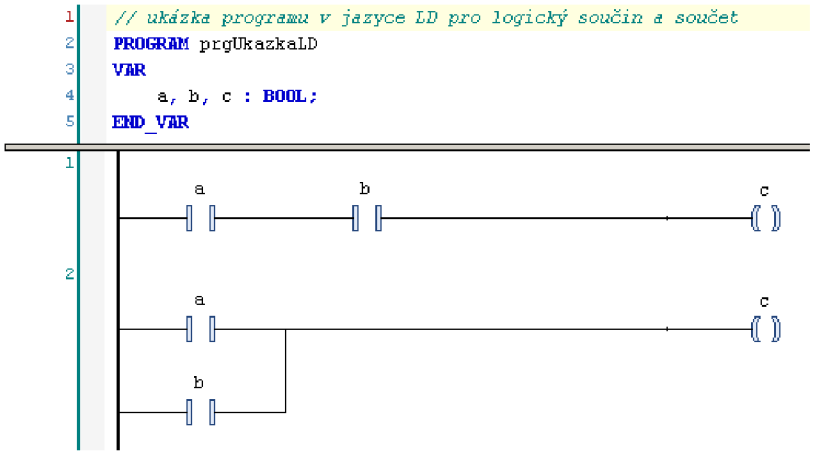
Syntaxe jazyku IL vzdáleně připomíná Asembler. Kód je psaný v instrukcích jako jsou např. `LD, ST, JMP, ADD, GT, CAL...`

```
// ukázka programu v jazyce IL pro součet dvou čísel
PROGRAM prgIL
  VAR
    a, b, c : INT;
  END_VAR

  LD  a
  ADD b
  ST  c
END_PROGRAM
```

Jazyk ST již připomíná některý z vyšších jazyků jako je Basic nebo Pascal. Při programování v ST jsou již používána klíčová slova jako `IF, FOR, CASE` a běžné operátory např. `+, := , >, NOT, AND, ...`

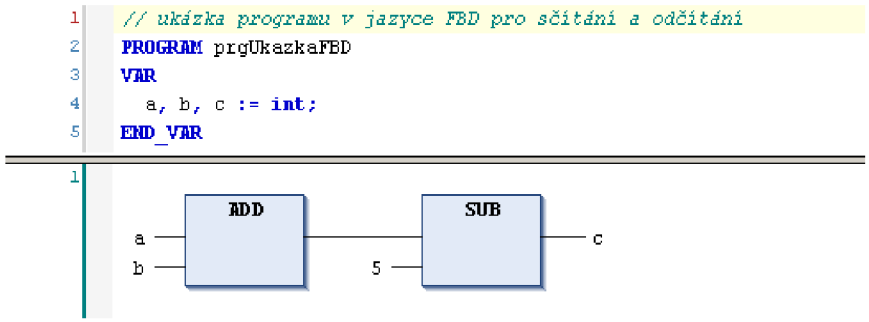
```
// ukázka programu v jazyce ST pro součet dvou čísel
PROGRAM prgST
  VAR
    a, b, c : INT;
  END_VAR

  c := a + b;
END_PROGRAM
```

Grafický jazyk LD je čitelný nejen programátorům PLC ale i všem lidem z technické praxe, kteří umějí pracovat s reléovými schématy. Programování v LD spočívá v propojení napěťových linií pomocí sítě spínacích a rozpínacích kontaktů reprezentujících vstupy na výstupy. LD je vhodné především pro programování logických obvodů.



Obrázek 1: Ukázka jazyka LD

V grafickém jazyku FBD se jednotlivé elementy propojují linkami představující signálový tok. Elementy potom představují například funkce nebo instance funkčních bloků.



Obrázek 2: Ukázka jazyka FBD

3.2.1 Nezávislé použití různých jazyků

Deklarační část proměnných POU je ve všech jazycích psána v textu a má stejnou syntaxi. Deklarační část slouží jako rozhraní POU, protože obsahuje všechny vstupní a výstupní proměnné. Díky tomu je možné v rámci jedné aplikace používat a vzájemně tvořit vazby mezi POU implementovaných v libovolném z jazyků. Například funkční blok psaný v jazyce ST může volat jiný funkční blok vytvořený v jazyce LD, protože jediné co pro volání potřebuje znát je deklarace vstupních a výstupních proměnných volaného bloku.

Pokud se na jednotlivé jazyky podíváme z pohledu programování zjistíme, že si nejsou úplně rovnocenné. Největší omezení existují v jazyce LD, který je určen především pro práci s bitovými proměnnými a operace s číselnými a jinými proměnnými v něm lze realizovat pouze voláním funkcí a funkčních bloků vytvořených v jiných jazycích. Na druhou stranu má nespornou výhodu právě v programování logických operací díky přehlednosti a čitelnosti výsledných schémat.

Jazyk ST programátorovi nabízí asi největší volnost, protože umí pracovat jak s bitovými tak i jinými datovými typy (čísla, text, čas) a má přímo v syntaxi podporu pro větvení programu či vytváření cyklů. O jazyku IL lze s ohledem na dokonalost většiny překladačů ST asi stále tvrdit, že je v něm možné vytvářet rychlejší a na kapacitu výkonu a paměti méně náročný kód než v jazyce ST. Ale o to více pracnější je vývoj aplikace.

3.3 Funkční blok z pohledu OOP

Pokud bychom ve druhé verzi normy IEC 61131-3 měli vybrat to, co se nejvíce podobá objektu v pojetí objektově orientovaného programování, pak by to byl funkční blok.

Jak již bylo zmíněno funkční blok má vnitřní paměť stavu a je instanční. Také je možné funkční blok považovat za zapouzdřený, protože k vnitřním proměnných funkčního bloku není z vnějšku umožněn přístup. Mezi funkčními bloky lze vytvářet vazby, kdy jeden funkční blok může být vstupem nebo výstupem jiného bloku. A také je možné vytvářet vazby, kdy jeden funkční blok obsahuje jiný.

Na rozdíl od objektu ale funkční blok neobsahuje žádné metody ani vlastnosti neumožňuje dědičnost, nemá podporu rozhraní a polymorfizmu.

3.4 Rozšíření o OOP

Jeden z návrhů pro třetí verzi normy IEC 61163-3 je právě rozšíření o prvky objektově orientovaného programování, viz [7] [8]. Snahou je rozšířit syntaxi konstrukce a použití funkčního bloku, tak aby se co nejvíce přiblížila koncepci tříd z OOP. Přestože zatím nebyla třetí verze normy vydána, začali již někteří tvůrci vývojových nástrojů toto rozšíření implementovat do svých prostředí pro programování PLC.

3.4.1 CoDeSys

CoDeSys (Controller Development System) je vývojové prostředí pro programování průmyslových automatů dle normy IEC 61131-3. Je vyvíjeno společností 3S-Smart Software Solutions GmbH, která se aktivně podílí na vývoji této normy. Slouží k programování různých typů PLC od různých výrobců, kteří nechtějí investovat do vývoje vlastního prostředí, a je tedy nezávislá na konkrétním hardwaru. Díky tomu se jedná o obecný a velmi propracovaný nástroj pro programování PLC. Prostředí navíc obsahuje různá rozšíření normy jako je například vlastní grafický programovací jazyk volných funkčních bloků CFC (Continuous Function Cart) a od verze 3 implementuje také rozšíření o objektově orientované programování.

3.5 Funkční blok jako třída

Tato kapitola se věnuje rozšíření funkčního bloku o prvky konceptu třídy z OOP tak, jak

je v současné době implementovaná právě prostředím CoDeSys, blíže viz [4]. V následujícím textu budu používat označení „původní“ pro funkční blok dle druhé verze normy.

Jak bylo nastíněno v předchozích kapitolách, jedním z předpokladů pro umožnění dědičnosti funkčního bloku je rozdělení jeho funkčnosti na více částí. Původní funkční blok je tedy rozšířen o možnost obsahovat metody a vlastnosti podobně, jako je tomu u tříd. V souvislosti s vlastní dědičností přibývají nová klíčová slova jako `EXTENDS`, nebo `SUPER`. Dále je doplněna i podpora rozhraní, referenčních typů nebo ukazatelů. Užitečná může být i nová možnost deklarace statických proměnných.

Rozšíření se týká především deklaračních částí funkčního bloku, které jsou vždy psány v jazyce ST. Pro kód vlastního těla bloku zůstává možnost implementace v libovolném z uvedených jazyků.

Zůstává však zachována možnost nadále používat původní funkční blok, tak jak byl definován v první a druhé verzi normy. To znamená, že programátor není nucen používat žádné prvky OOP a všechny předkladače musí být zpětně kompatibilní.

3.5.1 Metody

Metody na rozdíl od funkcí jsou svázané s konkrétním funkčním blokem. Deklarace metody obsahuje vstupní proměnné a typ návratové hodnoty. Tělo metody může obsahovat další proměnné, které jsou však platné pouze v době vykonávání metody. Dále je v těle metody umožněn přístup ke všem vnitřním proměnným funkčního bloku mimo vstupně-výstupních a dočasných proměnných. Pro přístup k vnitřním proměnným funkčního bloku je možno použít klíčové slovo `THIS`, které slouží jako identifikátor ukazatele (viz dále) na funkční blok. Například v případě, že vnitřní proměnná metody je stejně pojmenovaná jako proměnná funkčního bloku a zastiňuje ji.

Jednotlivé metody je možno psát v různých jazycích. To může být v některých případech velice užitečné. Například pokud daná metoda pracuje převážně s bitovými proměnnými, je možné pro ni zvolit jazyk LD a v jiné metodě, kde se provádí číselné výpočty, naopak použít jazyk ST.

```
// ukázka vytváření metod funkčního bloku
FUNCTION_BLOCK fbMetody
  VAR
    a, b, c : int;
  END_VAR

  METHOD Secti : int
    VAR_INPUT
      c : int;
    END_VAR

    THIS^.c := a + b + c;
    Secti := THIS^.c;
```

```
  END_METHOD
END_FUNCTION_BLOCK
```

3.5.2 Speciální metody

Pro funkční blok jsou deklarovány následující speciální funkce:

- inicializační metoda `FB_init`: obsahuje inicializační kód. Je deklarována implicitně, ale může být deklarována i explicitně. V případě explicitní deklarace se nejdříve vykoná implicitní a poté explicitní. Metoda může mít další vstupní proměnné. Inicializace se provádí například po nahrání programu do PLC nebo při jeho restartu.
- reinicializační metoda `FB_reinit`: obsahuje kód pro inicializaci nové instance funkčního bloku, který vznikl kopií jiného bloku, tak aby hodnoty původního bloku byly zkopírovány do nové instance. Ke kopírování bloků dochází např. při on-line změnách kódu. Metoda musí být deklarována i volána explicitně. Metoda nemá žádné vstupní proměnné.
- ukončovací metoda `FB_exit`: metoda musí být deklarována explicitně a je volána pro instance funkční bloků, které jsou přesunovány nebo mazány při změnách kódu.

Speciální metody slouží především pro systémové účely a dá se předpokládat, že jejich použití bude platformně závislé. Nicméně je možné je volat i uživatelsky jako každou jinou metodu, ale praktický význam bych viděl asi pouze pro inicializační metodu.

Při pohledu na tyto metody je možné najít jistý náznak podobnosti s konstruktory a destruktory z OOP. Konstruktory slouží k inicializaci nově vytvářeného objektu a destruktory naopak k jejich zrušení. V souvislosti s konstruktory a destruktory si je však nutné uvědomit rozdíl mezi statickým a dynamickým přístupem k vytváření objektů a přidělování paměti. Statický přístup znamená, že všechny proměnné jsou deklarovány již v době návrhu aplikace a překladač jim pevně přidělí místo v paměti. Za běhu aplikace již není možné vytvářet nové proměnné. Zatímco dynamický přístup umožňuje za běhu aplikace vytvářet nové proměnné (resp. rušit staré) a přidělovat jim novou paměť (resp. uvolňovat). Konstruktory a destruktory najdou uplatnění právě v dynamickém přístupu. Naopak norma IEC 61113-3 představuje statický přístup a jediný okamžik, kdy dochází ke změně rozložení paměti je se změnou aplikace.

3.5.3 Vlastnosti

Vlastnost může být přiřazena programu nebo funkčnímu bloku (POU). Vlastnost obsahuje dvě speciální metody:

- metoda `SET`, která slouží pro zápis dat do objektu, slouží jako jeho vstup,
- metoda `GET`, která slouží pro čtení dat objektu, slouží jako jeho výstup.

Vlastnost může obsahovat vlastní vnitřní proměnné, ale nesmí mít další vstupní ani výstupní proměnné. Vychází to z logiky, že metoda `SET` automaticky přijímá jednu vstupní hodnotu a metoda `GET` naopak vrací jednu výstupní hodnotu. Obě hodnoty jsou datového typu uvedeného v deklaraci vlastnosti.

```
// ukázka vytváření vlastností funkčního bloku
FUNCTION_BLOCK fbVlastnosti
  VAR
    a : bool;
  END_VAR
  PROPERTY Cislo : int
    GET
      Cislo := a;
    END_GET
    SET
      a := LIMIT(0, Cislo, 20);
    END_SET
  END_PROPERTY
END_FUNCTION_BLOCK
```

3.5.4 Rozhraní

Rozhraní slouží k popisu metod a vlastností, jaké musí obsahovat funkční blok, který dané rozhraní implementuje. Popisem metody nebo vlastnosti je myšlena pouze její deklarace, která neobsahuje žádnou implementaci, nemá žádné tělo ani vnitřní proměnné. Deklarace obsahuje název, vstupní, výstupní a vstupně-výstupní proměnné. Tělo metody musí být doplněno až pro konkrétní implementaci rozhraní daného funkčního bloku. Funkční blok může implementovat několik rozhraní současně.

```
// ukázka vytvoření a použití rozhraní
INTERFACE rSecti
  METHOD Pricti : real
    VAR_INPUT
      b : real;
    END_VAR
  END_METHOD
  PROPERTY Hodnota : real
    GET
    END_GET
  END_PROPERTY
END_INTERFACE

FUNCTION_BLOCK fbSectiInt IMPLEMENTS rSecti
  VAR
    a : int;
  END_VAR

  METHOD Pricti : real
    VAR_INPUT
```

```
      b : real;
    END_VAR
    a := a + REAL_TO_INT(b);
    Pricti INT_TO_REAL(a);
  END_METHOD
  PROPERTY Hodnota : real
    GET
      Hodnota := INT_TO_REAL(a);
    END_GET
  END_PROPERTY
END_FUNCTION_BLOCK

FUNCTION_BLOCK fbSectiReal IMPLEMENTS rSecti
  VAR
    d : real;
  END_VAR

  METHOD Pricti : real
    VAR_INPUT
      b : real;
    END_VAR
    d := d + b;
    Pricti d;
  END_METHOD
  PROPERTY Hodnota : real
    GET
      Hodnota := d;
    END_GET
  END_PROPERTY
END_FUNCTION_BLOCK
```

3.5.5 Dědičnost

Funkční blok lze odvodit od jiného funkčního bloku. Nový funkční blok automaticky obsahuje všechna data a metody původního bloku. Pro přístup k datům původního bloku slouží klíčové slovo `SUPER`, které podobně jako `THIS` slouží jako identifikátor ukazatele (viz dále) na blok. Metody původního bloku mohou být nahrazeny novou implementací tím, že v novém bloku budou deklarovány metody stejného názvu se stejnými vstupními a výstupními proměnnými. Nový blok může navíc obsahovat rozšíření o své vlastní metody a data. Vícenásobná dědičnost však není povolena. Dědičnost lze použít i pro rozhraní.

```
// ukázka dědičnosti
FUNCTION_BLOCK fbPocitani EXTENDS fbSectiInt
  VAR
    f : int;
  END_VAR
  METHOD Odecti : int
    VAR_INPUT
      b : int;
```

```
    END_VAR

    f := f - SUPER^.a - b;
    Odecti := f;
  END_METHOD

  SUPER^();
END_FUNCTION_BLOCK
```

3.5.6 Ukazatel

Ukazatel (pointer) je proměnná, která za běhu programu obsahuje adresu jiné proměnné, funkčního bloku nebo programu. Je možné vytvářet i ukazatele na funkce nebo metody, ale ty jsou určeny pouze pro systémové potřeby. Pro přístup k objektu, na jehož adresu míří ukazatel je nutné provést dereferenci ukazatele pomocí operátoru `^`. Naopak pro naplnění adresy slouží operátor `ADR`.

```
// ukázka použití ukazatele
PROGRAM prgUkazatel
  VAR
    a, b : int;
    PtrInt : pointer to int;
  END_VAR

  PtrInt := ADR(a);
  b := PtrInt^;
END_PROGRAM
```

3.5.7 Reference

Reference je alias pro objekt neboli přidání dalšího identifikátoru (názvu) pro jeden a ten samý objekt. Rozdíl mezi referencí a ukazatelem spočívá v tom, že při práci s referencí jsou všechny operace prováděné přímo se zastoupeným objektem, zatímco při práci s ukazatelem je nutná dereference. Pro přiřazení do reference je používán speciální operátor `ref=`. Proměnné typu rozhraní jsou automaticky považovány za reference.

```
// ukázka použití reference
PROGRAM prgReference
  VAR
    a, b : int;
    RefInt : reference to int;
  END_VAR

  RefInt ref= a;
  a := RefInt + 5;  // a := a + 5;
  RefInt ref= b;
  a := RefInt + 5;  // a := b + 5;
END_PROGRAM
```

3.5.8 Statické proměnné

Statické proměnné je možné použít ve funkčním bloku, metodě nebo funkci. Deklarují se pomocí klíčového slova `VAR_STAT`. Statické proměnné slouží jako sdílené proměnné pro všechny instance daného funkčního bloku. V případě použití ve funkci se chová podobně jako globální proměnná. Statické proměnné si pamatují svůj stav mezi jednotlivými voláními, lze je tedy použít např. jako čítač instancí daného bloku nebo čítač volání metody či funkce. Inicializace statické proměnné je provedena před prvním voláním. Přístup ke statické proměnné z vnějšku funkčního bloku je možný přes jeho libovolnou instanci.

```
// ukázka statických proměnných
FUNCTION_BLOCK fbStat
  VAR_STAT
    a : int := 0;
  END_VAR

  a := a + 1;
END_FUNCTION_BLOCK

PROGRAM prgStat
  VAR
    Stat1, Stat2 : fbStat;
  END_VAR

  Stat1();  // Stat1.a = Stat2.a = 1
  Stat2();  // Stat1.a = Stat2.a = 2
END_PROGRAM
```

3.5.9 Volání metod

Syntaxe volání metod je podobná volání funkcí. Jediný rozdíl je, že při volání metody je nutné pomocí tečkové konvence spojit metodu s konkrétní instancí funkčního bloku. Nadále zůstává možnost volat funkční blok jako takový.

```
// příklad volání metody instance funkčního bloku
PROGRAM prgVolaniMetod
  VAR
    Pocitani1 : fbPocitani;
  END_VAR

  // volání metody
  Pocitani1.Odecti(b := 5);

  // volání funkčního bloku
  Pocitani1();
END_PROGRAM
```

3.5.10 Polymorfizmus

Se zavedením možnosti používání rozhraní nebo dědičnosti přichází na řadu další prvek objektově orientovaného programování - polymorfizmus.

Rozhraní by bez polymorfizmu v podstatě ani nemělo význam. Právě při použití rozhraní deklarujeme, že je vytvářena vazba s dopředu neznámým funkčním blokem. Pouze je požadováno, aby měl jakýmkoli způsobem implementované definované metody a vlastnosti. Typ tohoto bloku bude určen až v konkrétní implementaci programu. Výsledná vazba potom může být statická, kdy se typ bloku za běhu programu nemění, nebo může být dynamická.

```
// ukázka polymorfizmu při použití rozhraní
PROGRAM prgPolymorfizmus1
  VAR
    Secti     : rSecti;
    SectiInt  : fbSectiInt;
    SectiReal : fbSectiReal;
    a : int;
  END_VAR

  Secti := SectiInt;
  a := Secti.Pricti(5); // volání metody funk. bloku fbSectiInt

  Secti := SectiReal;
  a := Secti.Pricti(5); // volání metody funk. bloku fbSectiReal
END_PROGRAM
```

Při rozšíření funkčního bloku pomocí dědičnosti je ve všech vazbách, kde byl očekáván rodičovský blok, možno použít blok potomka. Pokud je při takovéto vazbě volána některá z metod, která má v těle potomka jinou implementaci, bude přesto volána metoda rodiče, protože ten vystupuje v původní deklaraci vazby. Jiným způsobem by fungovalo volání přes ukazatel nebo referenci na funkční blok. Jedná se o případ, kdy je deklarovaná proměnná jako ukazatel nebo reference na rodičovský blok, ale je do ní přiřazen potomek. Potom při volání metody přes tuto proměnnou bude naopak volána metoda potomka.

```
// ukázka polymorfizmu při použití dědičnosti
PROGRAM prgPolymorfizmus2
  VAR
    TestRodic    : fbRodic;
    RefTestRodic : REFERENCE TO fbRodic;
    PtrTestRodic : POINTER TO fbRodic;

    Potomek : fbPotomek;
    Rodic   : fbRodic;
  END_VAR
```

```
  TestRodic := Rodic;
  TestRodic.Metoda();   // volání metody funk. bloku fbRodic
  TestRodic := Potomek;
  TestRodic.Metoda();   // volání metody funk. bloku fbRodic

  RefTestRodic ref= Rodic;
  RefTestRodic.Metoda();  // volání metody funk. bloku fbRodic
  RefTestRodic ref= Potomek;
  RefTestRodic.Metoda();  // volání metody funk. bloku fbPotomek

  PtrTestRodic := ADR(Rodic);
  PtrTestRodic^.Metoda(); // volání metody funk. bloku fbRodic
  PtrTestRodic := ADR(Potomek);
  PtrTestRodic^.Metoda(); // volání metody funk. bloku fbPotomek
END_PROGRAM
```

4 UDÁLOSTNĚ ŘÍZENÉ PROGRAMOVÁNÍ

Myšlenka událostně řízeného programování jednoduše řečeno spočívá v tom, že konkrétní část programu je vykonána pouze v okamžiku vzniku konkrétní události. Jako jedna z výhod použití událostně řízeného programování je tedy úspora výpočetního výkonu. Za událost může být považována jakákoli změna stavu např. změna stavu vstupu nebo uplynutí nastavené doby časovače. V [9] se jako typický příklad uvádí programování uživatelského rozhraní (Graphical User Interface - GUI). Tam jsou jako události vyhodnocovány operace prováděné uživatelem jako je například stisknutí tlačítka nebo zadání hodnoty.

Za událostní řízení by se dal považovat diagram vytvořený v SFC. Sekvence v SFC je rozdělena do jednotlivých kroků a těm jsou potom přiřazeny akce, které se vykonávají pouze během daného kroku. Akce, které jsou přiřazeny vzniku nebo zániku kroku, jsou potom vykonány jednorázově pro danou aktivaci kroku. V SFC lze tedy vytvořit program, který při vzniku události vykoná pouze příslušnou akci a v době mezi událostmi neprovádí nic.

Nicméně jádrem SFC je především řízení stavů. Stavy jsou pevně definovány a stejně tak i události, při kterých dojde k přechodu mezi nimi. Jádrem událostně řízeného programování nejsou stavy, ale naopak události. Princip se pokusím postupně vysvětlit na následujíc příkladu.

Pokud si představíme jednoduchý funkční blok, který zpracovává dva vstupy a má dva výstupy.

```
// funkční blok s dvěma vstupy a výstupy
FUNCTION_BLOCK fbPocty
  VAR_INPUT
    a, b : int;
  END_VAR
  VAR_OUTPUT
    c, d : int;
  END_VAR

  c := c + a;
  d := c + b;
END_FUNCTION_BLOCK
```

Je zřejmé, že jeho výstupní hodnoty se mohou změnit pouze v okamžiku, kdy se změní některý z jeho vstupů. Jindy by volaní tohoto funkčního bloku nemělo význam, protože by byl pouze znovu vypočítán ten samý výsledek, pouze by toto volání zabíralo procesorový čas.

Uvedený funkční blok lze přepsat tak, aby na začátku zjistil jestli došlo k nějaké změně vstupních hodnot a teprve po vyhodnocení této události provedl přepočet.

```
// funkční blok s podmíněním zpracování změnou vstupu
FUNCTION_BLOCK fbPocty
  VAR_INPUT
    a, b : int;
  END_VAR
  VAR
    a_pamet, b_pamet : int;
  END_VAR
  VAR_OUTPUT
    c, d : int;
  END_VAR

  IF a <> a_pamet or b <> b_pamet THEN
    a_pamet := a;
    b_pamet := b;
    c := c + a;
    d := c + b;
  END_IF;
END_FUNCTION_BLOCK
```

Na první pohled je vidět, že vyhodnocení události bude stát nějakou režii. V případě, že by některá vstupní hodnota byla použita i v jiné instanci, bylo vyhodnocení události prováděno opakovaně v každé z těchto instancí. Z hlediska úspory výkonu je výhodnější provést vyhodnocení události mimo funkční blok. V kódu se tedy vrátíme k první verzi funkčního bloku a vyhodnocení události provedeme na úrovni programu.

```
// program s podmíněním volání funkčních bloků změnou některého
// z jejich vstupů
PROGRAM prgUdalosti
  VAR_INPUT
    a, b : int;
  END_VAR
  VAR
    Pocty1, Pocty2 : fbPocty;
    a_pamet, b_pamet : int;
  END_VAR
  IF a <> a_pamet THEN
    a_pamet := a;
    Pocty1(a, b);
    Pocty2(a, b);
  END_IF;
  IF b <> b_pamet THEN
    b_pamet := b;
    Pocty1(a, b);
    Pocty2(a, b);
  END_IF;
END_PROGRAM
```

Když se podíváme na kód funkčního bloku `fbPocty`, zjistíme, že výsledek první rovnice `c := c + a` se změní pouze v případě, že se změnila hodnota prvního vstupu `a`. V případě změny pouze druhého ze vstupů nemusí být první rovnice přepočítána.

Druhá rovnice ale zpracována být musí. Vzniká tak požadavek na rozdělení kódu funkčního bloku na dvě samostatně vykonávané části. Pokud bychom použily větvení dostaly bychom se do podobné situace jako v případě vyhodnocování události uvnitř bloku. Na řadu přichází využití rozšíření funkčního bloku o prvky objektově orientovaného programování, konkrétně použití metod.

```
// rozdělení kódu funkčního bloku do metod
FUNCTION_BLOCK fbPocty
  VAR_INPUT
    a, b : int;
  END_VAR
  VAR_OUTPUT
    c, d : int;
  END_VAR

  METHOD PrictiA
    c := c + a;
    PrictiB();
  END_METHOD

  METHOD PrictiB
    d := c + b;
  END_METHOD
END_FUNCTION_BLOCK

PROGRAM prgUdalosti
  VAR_INPUT
    a, b : int;
  END_VAR
  VAR
    Pocty1, Pocty2 : fbPocty;
    a_pamet, b_pamet : int;
  END_VAR
  IF a <> a_pamet THEN
    a_pamet := a;
    Pocty1.a := a;
    Pocty1.PrictiA();
    Pocty2.a := a;
    Pocty2.PrictiA();
  END_IF;
  IF b <> b_pamet THEN
    b_pamet := b;
    Pocty1.b := b;
    Pocty1.PrictiB();
    Pocty2.b := b;
    Pocty2.PrictiB();
  END_IF;
END_PROGRAM
```

Pokud se podíváme blíže na program `prgUdalosti` provádějící vyhodnocení vzniku události, zjistíme, že provádí stále dokola dvě základní operace. Jedna operace je

vyhodnocení vzniku nové události jako změny stavu některé proměnné. Druhá operace je pak postupné volání metod, které mají na danou událost reagovat.
Priorita, s jakou jsou vyhodnocovány jednotlivé události, je dána pořadím jejich zpracování v programu. Pokud by byl požadavek na chronologické zpracování událostí, muselo by jejich vyhodnocování probíhat na pozadí. Vzniklé události by se za sebe řadily do fronty ke zpracování. Z této fronty by je potom odebíral program a rozesílal, ke zpracování jednotlivým metodám.

Ve výše uvedeném příkladě byly obě vyhodnocované události globální, vyhodnocovala se změna globálních proměnných. Události ale mohou vznikat i uvnitř funkčního bloku, kam nemá program přístup. V tom případě by vznik události musel být zapsán na výstup bloku a teprve potom zpracován programem. Program by tak musel kromě globálních událostí ještě neustále testovat všechny instance bloků na vznik jejich vnitřních událostí.

```
// vyhodnocení vzniku události uvnitř funkčního bloku
FUNCTION_BLOCK fbPocty
  VAR_INPUT
    a : int;
  END_VAR
  VAR
    c : int;
  END_VAR
  VAR_OUTPUT
    udalost_c : bool;
  END_VAR

  METHOD PrictiA
    c := c + a;
    udalost_c := true;
  END_METHOD
END_FUNCTION_BLOCK

PROGRAM prgUdalosti
  VAR
    Pocty : fbPocty;
  END_VAR

  IF Pocty.udalost_c THEN
    Pocty.udalost_c := false;
    ...
  END_IF;
END_PROGRAM
```

Určitě by bylo jednodušší, kdyby instance funkčního bloku při vzniku vnitřní události sama volala metody, které danou událost zpracovávají. Problém je, že metody, které má instance funkčního bloku volat, mohou být pro každou instanci různé a nejde je určit dopředu, ale vyplynou až z celkové implementace programu. Musel by se použít

mechanizmus, který je v anglické literatuře označován jako „Publish/Subscribe“, kdy se naopak jednotlivé metody zpětně registrují jako příjemci události konkrétní instance funkčního bloku. Každá instance by si tedy vedla svůj vlastní seznam zaregistrovaných příjemců a při vzniku události by je všechny zavolala.

Ve vyšších jazycích je pro události udělána podpora v syntaxi. Prakticky se to provádí pomocí dynamické tabulky ukazatelů na metody, které se mají při dané události vykonávat. Příjemci se do tabulky sami registrují. V případě normy žádná podpora na úrovní jazyka není a nezbývá než si vše obsloužit sám s využitím rozhraní. Výsledkem je mnoho kódu, který může výsledný program udělat hůře čitelný.

```
// příklad registrace příjemců událostí funkčního bloku
INTERFACE rUdalost_c
  METHOD PrijemUdalosti_c : bool
    VAR_INPUT
      c : real;
    END_VAR
    ...
  END_METHOD
END_INTERFACE

FUNCTION_BLOCK fbPocty
  VAR_INPUT
    a : int;
    Prijemci_c : array[0..9] of rUdalost_c;
    PrijemciPocet : int;
  END_VAR
  VAR
    c : int;
    i : int;
  END_VAR

  METHOD PrictiA
    c := c + a;
    FOR i := 1 TO  PrijemciPocet DO
      rUdalost_c.PrijemUdalosti_c(c);
    END_FOR;
  END_METHOD
END_FUNCTION_BLOCK

FUNCTION_BLOCK fbPrijemceUdalosti_c IMPLEMENTS rUdalost_c
  METHOD PrijemUdalosti_c : real
    VAR_INPUT
      c : real;
    END_VAR
    ...
  END_METHOD
END_FUNCTION_BLOCK

PROGRAM prgUdalosti
  VAR
```

```
    Prijemce1, Prijemce2 : fbPrijemceUdalosti;
    Pocty : fbPocty :=
    (Prijemci_c := [Prijemce1, Prijemce2],
    PrijemciPocet := 2);
  END_VAR

  Pocty.PrictiA(5);
END_PROGRAM
```

V příkladech byla použita událost, kde příčinou jejího vzniku byla změna vnitřního stavu následkem volání některé z metod funkčního bloku. V rámci volání metody se provede jak změna stavu, tak i vyhodnocení události. Jak ale řešit situaci, kdy událost vzniká uplynutím nastavené doby časovače použitého uvnitř funkčního bloku. V tom případě musí být daný časovač nepřetržitě zpracováván a jeho uplynutí vyhodnoceno. Pro zpracování časovačů máme dvě možnosti. Buď to budeme provádět mimo funkční blok a po uplynutí času zavoláme příslušnou metodu funkčního bloku. Nebo bude zpracování časovače uvnitř funkčního bloku a potom musíme zajistit jeho nepřetržité volání.

```
// vyhodnocení události s časovače použitého uvnitř funkčního
// bloku
FUNCTION_BLOCK fbCasVne
  VAR_INPUT
    Casovac : TON := (PT := t#10s);
  END_VAR
  METHOD Start : bool
    Casovac.IN := true;
  END_METHOD
  METHOD Stop : bool
    Casovac.IN := false;
  END_METHOD
END_FUNCTION_BLOCK

FUNCTION_BLOCK fbCasUvnitr
  VAR
    Casovac : TON := (PT := t#10s);
  END_VAR
  METHOD Start : bool
    Casovac.IN := true;
  END_METHOD
  METHOD Stop : bool
    Casovac.IN := false;
  END_METHOD

  Casovac();
  IF Casovac.Q THEN
    Stop();
  END_IF;
END_FUNCTION_BLOCK
```

```
PROGRAM prgUdalostiCas
  VAR
    CasVne    : fbCasVne;
    CasUvnitr : fbCasUvnitr;
  END_VAR

  // spuštění časovače
  IF ... THEN
    CasVne.Start();
    CasUvnitr.Start();
  END_IF;

  // zpracování a vyhodnocení uplynutí doby
  CasVne.Casovac();
  IF CasVne.Casovac.Q THEN
    CasVne.Stop();
  END_IF;

  CasUvnitr();
END_PROGRAM
```

Je vidět, že událostní řízení je možné. Je nutné rozdělit kód do metod, které budou zpracovávat jednotlivé události. Propojit zdroj události s jejím příjemcem staticky nebo „dynamicky“ pomocí rozhraní. V současnosti není žádná podpora na úrovni jazyka.

5 OBJEKTOVĚ ORIENTOVANÉ PROGRAMOVÁNÍ PRAKTICKY

V této kapitole bych chtěl projít hlavní části rozšíření normy o objektově orientované programování a na příkladech poukázat na výhody a úskalí jejich použití.

Ukázkový příklad aplikace vychází z jednoduchého funkčního bloku `fbMotor` napsaného dle druhé verze normy. Vlastní kód bloku je natolik zjednodušen, je určen více k demonstraci než k praktickému použití. Funkční blok obsluhuje jeden výstup na spuštění motoru. Jako zpětná hláška o chodu se mu vrazí signál o sepnutí stykače. Je možné přepínat režim ovládání motoru mezi ručním a automatickým. V ručním režimu je spouštěn tlačítkem, v automatickém režimu je spouštěn ve vazbě na konkrétní technologický proces.

```
// demonstrační příklad funkčního bloku motoru dle druhé verze
// normy
FUNCTION_BLOCK fbMotor
  VAR_INPUT
    iStartRuc,
    iStartAut,
    iRezimAut,
    iChod : bool;
  END_VAR
  VAR
    vProdlevaChod : TON := (PT := T#5S);
  END_VAR
  VAR_OUTPUT
    oStart,
    oPorucha : bool;
  END_VAR

  // vyhodnocení podmínky spuštění motoru
  oStart := not iRezimAut and iStartRuc or
                iRezimAut and iStartAut;

  // vyhodnocení poruchy chodu
  vProdlevaChod(IN := oStart xor iChod,
                Q => oPorucha);
END_FUNCTION_BLOCK
```

5.1 Metody

Funkční blok `fbMotor` obsahuje dvě logické rovnice. První z nich je vyhodnocení podmínky spuštění motoru. Druhá z nich je kontrola zpětné hlášky o sepnutí stykače a vyhodnocení poruchy chodu. Z pohledu objektového programování se jedná o jednotlivé části funkčnosti a zabalíme je do metod.

```
// rozdělení kódu funkčního bloku do metod
FUNCTION_BLOCK fbMotorOOP
  VAR_INPUT
    iStartRuc,
    iStartAut,
    iRezimAut,
    iChod : bool;
  END_VAR
  VAR
    vProdlevaChod : TON := (PT := T#5S);
  END_VAR
  VAR_OUTPUT
    oStart,
    oPorucha : bool;
  END_VAR
  METHOD Spusteni
    // vyhodnocení podmínky spuštění motoru
    oStart := not iRezimAut and iStartRuc or
                  iRezimAut and iStartAut;
  END_METHOD

  METHOD Porucha
    // vyhodnocení poruchy chodu
    vProdlevaChod(IN := oStart xor iChod,
                  Q => oPorucha);
  END_METHOD
END_FUNCTION_BLOCK
```

V takto jednoduchém případě není složité rozhodnout, jak rozdělit kód bloku do jednotlivých metod. U větších bloků to již tak zřejmé být nemusí. Volbu metod můžeme provést z pohledu rozdělení funkčnosti, kterou se snažíme rozdělit až na nejmenší již dále nedělitelné části. Tím získáme dostatečný prostor i pro využití následné dědičnosti. Na rozdíl od vyšších programovacích jazyků však můžeme každou metodu psát v jiném jazyce. Rozdělení metod potom může záviset na typu úlohy. Například pokud se v některé části bloku objevuje sekvenční řízení, můžeme tuto část zabalit to metody a naprogramovat ji pomocí SFC. Naopak pro logické operace využijeme metodu psanou v LD.

Jiný přístup by se zvolil při použití konceptu událostního programování. V tom případě by metody plnily funkci reakce na události. Návrh takového bloku by mohl vycházet v prvním kroku z určení jeho možných stavů a podmínek přechodů meni nimi. Ve druhém kroku potom vytvoření metod provádějící přechod mezi stavy při události vyvolané změnou příslušných podmínek.

5.1.1 Akce

CoDeSys již ve své druhé verzi zavedl akce. Jedná se v podstatě o stejný princip jako u metod jenom s tím rozdílem, že pro akce není možné deklarovat žádné vstupní nebo výstupní proměnné. Akce mohou pracovat pouze s proměnnými samotného funkčního

bloku. V našem příkladě `fbMotorOOP`, používáme metody bez vlastních proměnných a mohli bychom tedy místo nich využít akce. Metody jsou však obecnější než akce a je možné je použít namísto akcí, ale naopak to možné není. Zajímavé je, že v rámci dědičnosti je možné přepsat akci metodou. Nicméně v ukázkách kódu nebudu akce používat.

5.1.2 Volání funkčního bloku a metod

Podívejme se nyní na volání instance našeho funkčního bloku `fbMotorOOP` po zavedení metod.

```
// program volající instanci funkčního bloku
PROGRAM prgMotor
  VAR
    Motor : fbMotorOOP;
  END_VAR

  Motor(iRezimAut := false,
        iStartAut := false,
        iStartRuc := false,
        iChod     := Motor.oStart);

END_PROGRAM
```

První rozdíl, na který při zabalením funkčnosti do metod narazíme, je že při volání instance funkčního bloku není vykonán žádný jeho kód, resp. není volána žádná z jeho metod. První možností je doplnit do funkčního bloku volání jeho vlastních metod.

```
// doplnění těla funkčního bloku o volání vlastních metod
FUNCTION_BLOCK fbMotorOOP
  VAR_INPUT
    ...
  END_VAR

  METHOD Spusteni
    ...
  END_METHOD
  METHOD Porucha
    ...
  END_METHOD

  Spusteni();  // vykonání vlastní metody při volání bloku
  Porucha();   // vykonání vlastní metody při volání bloku

END_FUNCTION_BLOCK
```

Druhou možností je nepoužívat volání funkčního bloku, ale místo toho postupně volat přímo jeho metody. Tím zaprvé dojde k prodloužení zápisu a dále pak narazíme na

komplikaci s předáváním vstupních hodnot. Ty můžeme nejdříve naplnit do funkčního bloku a teprve potom volat jeho metody.

```
// program volající metody instance funkčního bloku
PROGRAM prgMotor
  VAR
    Motor : fbMotorOOP;
  END_VAR

  Motor.iRezimAut := false;
  Motor.iStartAut := false;
  Motor.iStartRuc := false;
  Motor.iChod     := Motor.oStart;

  Motor.Spusteni();
  Motor.Porucha();
END_PROGRAM
```

Nebo je možné rozšířit metody o vstupní proměnné a přes ně předávat hodnoty do funkčního bloku až při volání konkrétní metody. Tady však můžeme narazit na opakované předávání stejné vstupní hodnoty. Např. pokud vznikne požadavek na úpravu vyhodnocení poruchy motoru pouze v automatickém režimu. Tím pádem bude nutné předávat hodnotu vstupu `iRezimAut` jak do metody `Spusteni`, tak i do nově upravené metody `Porucha`.

```
// rozšíření metod funkčního bloku o vstupní proměnné
FUNCTION_BLOCK fbMotorOOP
  VAR
    vProdlevaChod : TON := (PT := T#5S);
  END_VAR
  VAR_OUTPUT
    oStart,
    oPorucha : bool;
  END_VAR
  METHOD Spusteni
    VAR_INPUT
      iRezimAut,
      iStartRuc,
      iStartAut : bool;
    END_VAR
    // vyhodnocení podmínky spuštění motoru
    oStart := not iRezimAut and iStartRuc or
                  iRezimAut and iStartAut;
  END_METHOD

  METHOD Porucha
    VAR_INPUT
      iChod,
      iRezimAut : bool;
    END_VAR
```

```
    // vyhodnocení poruchy chodu
    vProdlevaChod(IN := (oStart xor iChod) and iRezimAut,
                  Q => oPorucha);
  END_METHOD
END_FUNCTION_BLOCK

PROGRAM prgMotor
  VAR
    Motor : fbMotorOOP;
  END_VAR

  Motor.Spusteni(iRezimAut := false,
                 iStartAut := false,
                 iStartRuc := false);
  Motor.Porucha(iRezimAut := false,          // opakování vstupu
                iChod     := Motor.oStart);
END_PROGRAM
```

Jak bylo ukázáno, tak při zabalení funkčnosti bloku do metod je potřeba jejich volání uvnitř vlastního bloku. Druhá možnost volání jednotlivých metod z vně bloku vede na prodloužení a zesložitění zápisu. Jiná situace by samozřejmě nastala pokud by byl použit koncept událostně řízeného programování.

5.2 Vlastnosti

Funkční blok pro svou činnost přijímá a poskytuje data z vnějšku. Ve druhé verzi normy se jedná o vstupní, výstupní nebo vstupně-výstupní proměnné. Norma zároveň určuje, že hodnotu vstupních proměnných není možné měnit uvnitř funkčního bloku a naopak hodnotu výstupních proměnných není možno měnit vně bloku. Pouze vstupně-výstupní proměnné je možné měnit jak uvnitř, tak i vně funkčního bloku a je pro ně použita nepřímá adresace (přístup přes ukazatel). Druhá verze normy umožňuje pouze přímý přístup ke těmto proměnným.

Ve třetí verzi je doplněna možnost přístupu prostřednictvím vlastností, které již obsahují kód pro zpracování předávaných dat. Zpracováním může být nějaký přepočet, kontrola rozsahu, ... V rámci vlastnosti lze jednoduše implementací metod `GET` nebo `SET` určit jestli se bude jednat o vstupní, výstupní nebo vstupně-výstupní data.

```
// doplnění vlastnosti funkčního bloku
FUNCTION_BLOCK fbMotorOOP
  VAR_INPUT
    ...
  END_VAR
  VAR
    vProdlevaChod : TON := (PT := T#5S);
  END_VAR
  VAR_OUTPUT
    ...
  END_VAR
```

```
  ...

  PROPERTY PoruchaProdleva : time
    GET
      PoruchaProdleva := vProdlevaChod.PT;
    END_GET
    SET
      vProdlevaChod.PT := LIMIT(t#0s, PoruchaProdleva, t#30s);
    END_SET
  END_PROPERTY

END_FUNCTION_BLOCK
```

Vlastnosti však nelze využít např. při zadávání parametrů obsluhou z vizualizace, protože ovladače komunikace s PLC ve většině případů přistupují přímo k hodnotám v paměti a neumožňují vzdálené volání metod. Případné kontroly nebo úpravy rozsahů takto zadávaných hodnot parametrů musí být provedeny buď na úrovni vizualizace nebo lépe přímo ve funkčním bloku v PLC před každým jeho zpracováním.

5.3 Dědičnost

Dle druhé verze normy se funkční blok chová jako by měl pouze jednu jedinou metodu, která má jako své argumenty vstupní proměnné bloku a která se vykoná při volání bloku. Implementací této metody je celý kód obsažený ve funkčním bloku. Návratové hodnoty metody jsou potom výstupní proměnné bloku.

Pokud bychom se zamysleli nad možností dědičnosti funkčního bloku, tak bychom si museli položit otázku, co bychom tím vlastně získali. Dědičnost, jak již bylo zmíněno slouží k rozšíření funkčnosti objektu. Rozšíření znamená, že nějaká část původní funkčnosti bude zachována a zbylá část bude změněna. Nebo bude celá původní funkčnost zachována a bude přidána další nová funkčnost. Funkční blok však obsahuje pouze jednu jedinou metodu, to znamená, že nelze ji částečně zachovat a částečně upravit. Lze ji pouze zachovat celou nebo celou změnit. V přídě, že ji celou změníme, nelze potom hovořit o dědičnosti.

Zůstává tedy pouze možnost celou funkčnost zachovat v nezměněné podobě a poté ji rozšířit. Tuto koncepci by bylo možné realizovat tak, že by blok potomka obsahoval a volal blok rodiče. Vlastní rozšíření by bylo možno vložit před nebo za volání bloku rodiče. Tato možnost je však velice omezená. Často bychom chtěli změnit jen malou část logiky, ale protože je její výsledek použit i ve zbytku logiky, musíme ve výsledku velkou část logiky zkopírovat mimo blok nebo tvořit různé nečitelné konstrukce, abychom docílili byť i drobné změny.

```
// jeden z možných způsobů rozšíření funkčního bloku dle druhé
// verze normy
FUNCTION_BLOCK fbMotor2
```

```
  VAR_INPUT
    iStartRuc,
    iStartAut,
    iRezimAut,
    iChod,
    iTepOchrana : bool;
  END_VAR
  VAR
    vMotor : fbMotor;
  END_VAR
  VAR_OUTPUT
    oStart,
    oPorucha : bool;
  END_VAR

  // volání bloku rodiče
  vMotor(iStartRuc := iStartRuc,
         iStartAut := iStartAut,
         iRezimAut := iRezimAut,
         iChod     := iChod,
         oStart    => oStart);

  // rozšíření o tepelnou ochranu
  oPorucha := vMotor.oPorucha or iTepOchrana;

END_FUNCTION_BLOCK
```

V souvislosti druhé verze normy nelze tedy o dědičnosti hovořit. Drobné změny nebo úpravy logiky funkčního bloku je možné řešit pomocí různých vstupních parametrů, které slouží jako přepínače mezi jednotlivými variantami. Tím se však výsledný kód stává méně čitelný. Větší zásahy pak vyžadují zkopírování celého původního kódu a jeho následnou úpravu. Do problému se programátor dostane v okamžiku, kdy zjistí, že potřebuje upravit původní kód bloku rodiče a neuvědomí si, že tato část byla zachována i v některých jeho potomcích a musí ji tedy změnit na více místech v programu. Vzniká tak špatně udržovatelný kód náchylný k tvorbě chyb.

Rozšíření normy o dědičnost velmi výrazně zvyšuje možnosti znovupoužití kódu. Z tohoto pohledu se dědičnost jeví jako nejvýznamnější prvek, který rozšíření normy přináší. Zavedení metod bylo podmínkou dědičnosti. Vzhledem k omezeným možnostem použití událostního programování vyvstává otázka využití metod z jiného důvodu než je právě dědičnost.

```
// rozšíření funkčního bloku využitím dědičnosti
FUNCTION_BLOCK fbMotorOOP2 EXTENDS fbMotorOOP
  VAR_INPUT
    iTepOchrana : bool;
  END_VAR

  Super^();
  oPorucha := oPorucha or iTepOchrana;
```

```
END_FUNCTION_BLOCK
```

Při dědičnosti funkčního bloku je důležité si uvědomit, že tělo funkčního bloku je vlastně také metoda, implicitní, hlavní, která je vykonána při volání funkčního bloku. A už jenom pouhým faktem, že se provede dědění je tato hlavní metoda automaticky nahrazena novou, která je prázdná. Pokud nechceme hlavní metodu rodiče měnit, musíme provést její volání z těla potomka pomocí `Super^()`[1].

Zaměřme se nyní na vlastní rozšíření funkčního bloku a jeho umístění. Funkční blok `fbMotorOOP` jsme rozšířili o vyhodnocení tepelné ochrany jako poruchy motoru. Vlastní zpracování jsme umístily přímo do těla funkčního bloku. To ale není správně. Důvodem je, že kdybychom stejným způsobem přidaly rozšíření dalších jeho funkcí a všechny je umístily do těla, uzavřely bychom si možnost dalšího dědění potomka. Například pokud bychom novým děděním chtěli provést změnu ve vyhodnocení tepelné ochrany, nemohli bychom již volat tělo rodiče pomocí `Super^()`, ale museli bychom jej opsat a upravit pouze část vyhodnocení ochrany. Správně je tedy rozšířit metodu vyhodnocení poruchy.

```
// přesun kódu rozšíření do příslušné metody
FUNCTION_BLOCK fbMotorOOP2 EXTENDS fbMotorOOP
  VAR_INPUT
    iTepOchrana : bool;
  END_VAR
  METHOD Porucha
    Super^.Porucha();
    oPorucha := oPorucha or iTepOchrana;
  END_METHOD

  Super^();
END_FUNCTION_BLOCK
```

Opět je důležité si uvědomit, co se stane při volání těla rodiče pomocí `Super^()`. Tělo rodiče obsahuje volání své metody `Porucha`. Tu jsme ale u jeho potomka přepsaly a díky tomu bude volána tato metoda potomka. Pokud by tak tomu nebylo a z těla rodiče by byla volána původní metoda rodiče, nemohli bychom volání těla rodiče použít a museli bychom jej na úrovni potomka přepsat.

1 Je zvláštní, že pro přístup k rodiči je využíván ukazatel. Jednodušší a čitelnější by bylo použít referenci, kde by odpadla opakovaná nutnost zápisu dereference.

5.3.1 Vícenásobná a postupná dědičnost

Vícenásobná dědičnost, kdy potomek má více rodičů není povolena. Při postupné dědičnosti z potomka na potomka je omezení v přístupu na pouze nejbližšího rodiče. Není možné přistoupit k některému z prarodičů např. zápisem `Super^.Super^`. Problém by tak nastal v okamžiku, kdyby se potomek chtěl vrátit k implementaci metody prarodiče, která byla na úrovni rodiče přepsána.

```
// rozšíření funkčního bloku děděním do další generace
FUNCTION_BLOCK fbMotorOOP3 EXTENDS fbMotorOOP2
  VAR_INPUT
    iOtacky : bool;
  END_VAR
  VAR
    vProdlevaOtacky : TON := (PT := t#20s);
  END_VAR
  METHOD Porucha
    SUPER^.Porucha();
    vProdlevaOtacky(In := iOtacky xor oStart);
    oPorucha := oPorucha or vProdlevaOtacky.Q;
  END_METHOD

  Super^();
END_FUNCTION_BLOCK
```

5.4 Rozhraní

Pokud bychom se na funkční blok dívali z pohledu druhé verze normy, tak právě vstupní a výstupní proměnné tvořili jeho rozhraní. Ale v rámci rozhraní jako rozšíření o OOP není možné deklarovat žádné proměnné funkčního bloku, pouze metody nebo vlastnosti. Nelze tedy deklarovat, jaké proměnné musí obsahovat funkční blok, který implementuje dané rozhraní. Nelze rozhraní použít pro funkční blok bez metod nebo vlastností.

```
// rozhraní
INTERFACE iMotor
  METHOD Spusteni
  END_METHOD
  METHOD Porucha
  END_METHOD
END_INTERFACE

FUNCTION_BLOCK fbMotorOOP IMPLEMENTS iMotor
  ...
END_FUNCTION_BLOCK

PROGRAM prgMotor
  VAR
```

```
    Motor    : fbMotorOOP;
    Rozhrani : iMotor;
  END_VAR

  Rozhrani.Spusteni(); // volání před přiřazením způsobí
                       // run-time chybu !
  Rozhrani := Motor;

  Rozhrani.Spusteni();
  Rozhrani.Porucha();

  // Rozhrani(); // samotné rozhraní nelze volat !
END_PROGRAM
```

Na konceptu rozhraní se asi nejvíce projeví dualita pojetí funkčního bloku podle druhé verze normy a podle OOP. Při klasickém volání funkčního bloku nelze rozhraní využít, protože samotné rozhraní nelze volat, pouze jeho metody nebo vlastnosti. Volání rozhraní není možné např. z důvodu, že volání konkrétního funkčního bloku může obsahovat nutné přiřazení vstupně-výstupních proměnných. V rozhraní se nedeklarují žádné proměnné a v okamžiku překladu tedy není možné zjistit, jaké vstupně-výstupní proměnné by museli při volání být naplněny.

Možností jak obejít toto omezení je zabalení kódu v těle funkčního bloku a všech jeho vstupně-výstupních[2] proměnných do samostatné metody. Tu potom definovat v rozhraní.

```
// zabalení kódu v těle funkčního bloku do metody
INTERFACE iMotor
  METHOD Hlavni
    VAR_IN_OUT
      ioPoruchaGlob : bool;
    END_VAR
  END_METHOD
  METHOD Spusteni
  END_METHOD
  METHOD Porucha
  END_METHOD
END_INTERFACE

FUNCTION_BLOCK fbMotorOOP IMPLEMENTS iMotor

  ...

  METHOD Hlavni
    VAR_IN_OUT
      ioPoruchaGlob : bool;
    END_VAR
```

2 Přestože je v nápovědě prostředí CoDeSys uvedeno, že metoda nemá přístup k proměnným VAR_IN_OUT funkčního bloku (nejspíše opět z důvodu jejich naplnění před použitím), překladač vše přeloží v pořádku.

```
    This^(ioPoruchaGlob := ioPoruchaGlob);
    ioPoruchaGlob := ioPoruchaGlob or oPorucha;
  END_METHOD

  Spusteni();
  Porucha();
END_FUNCTION_BLOCK

PROGRAM prgMotor
  VAR
    Motor    : fbMotorOOP;
    Rozhrani : iMotor;
    PoruchaGlob : bool;
  END_VAR

  Rozhrani := Motor;
  Rozhrani.Hlavni(ioPoruchaGlob := PoruchaGlob);
END_PROGRAM
```

5.5 Polymorfizmus

Polymorfizmus umožňuje použití funkčního bloku potomka všude tam, kde byl očekáván blok rodiče. Například v předchozím příkladě můžeme provést změnu typu instance motoru z `fbMotorOOP` na `fbMotorOOP3`.

```
// polymorfizmus
PROGRAM prgMotor
  VAR
    Motor    : fbMotorOOP3;
    Rozhrani : iMotor;
    PoruchaGlob : bool;
  END_VAR

  Rozhrani := Motor;
  Rozhrani.Hlavni(ioPoruchaGlob := PoruchaGlob);
END_PROGRAM
```

Přiřazení typu `fbMotorOOP3` do rozhraní typu `iMotor` není problém, protože rozhraní bylo implementováno typem `fbMotorOOP`, ze kterého byl `fbMotorOOP3` postupně děděn. Při volání metoty `Hlavni` je volána metoda typu `fbMotorOOP`, protože v žádném z jeho potomků nebyla tato metoda přepsána. Metoda `Hlavni` volá samotný funkční blok, v našem případě je voláno tělo bloku `fbMotorOOP3`. V těle bloku `fbMotorOOP3` je voláno tělo bloku jeho předchůdce, ze kterého je voláno tělo bloku `fbMotorOOP`. Takže poměrně složitou cestou se opět dostáváme do těla bloku `fbMotorOOP`, ve kterém jsou volány metody `Spusteni` a `Porucha`. Metoda `Porucha` je ale přepsána v bloku potomka `fbMotorOOP3` a proto bude vykonána. Z této metody je postupně volána metoda `Porucha` rodiče `fbMotorOOP2` a z ní metoda `Porucha`

opět bloku `fbMotorOOP`. Následně je v bloku `fbMotorOOP2` porucha rozšířena o vyhodnocení poruchy tepelné ochrany a v bloku `fbMotorOOP3` o vyhodnocení poruchy otáček.

Výsledkem je, že v místě kde byl použit blok, který pouze vyhodnocoval poruchu chodu, jsme použili blok, který kromě poruchy chodu vyhodnocuje navíc poruchy tepelné ochrany a otáček. A to pouze změnou v deklaraci typu proměnné bez nutnosti měnit výkonný kód.

5.6 Reference

Reference umožňuje zápis operací s nepřímým přístupem k objektu (ukazatel) jako by se jednalo o objekt s přímým přístupem (není nutný symbol dereference). Pro přiřazení do proměnné typu reference byl vytvořen nový operátor `ref=`.[3]

Pokud pominu používání ukazatelů, tak reference by mohli nahradit vstupně-výstupní proměnné funkcí a funkčních bloků. Reference je možné použít jako vstupní nebo vnitřní proměnné. Není je možné použít jako výstupní proměnné. Nejspíš protože by výstupní reference musely být přiřazeny před vlastním voláním funkce nebo bloku.

```
// reference
FUNCTION_BLOCK fbMotorOOP
  VAR_INPUT
    iStartRuc,
    iStartAut,
    iRezimAut,
    iChod : bool;
    refPoruchaGlob : reference to bool;
  END_VAR
  VAR
    vProdlevaChod : TON := (PT := T#5S);
  END_VAR
  VAR_OUTPUT
    oStart,
    oPorucha : bool;
  END_VAR

  METHOD Porucha
    // vyhodnocení poruchy chodu
    vProdlevaChod(IN := (oStart xor iChod) and iRezimAut,
                  Q => oPorucha);
    refPoruchaGlob := refPoruchaGlob or oPorucha;
  END_METHOD

  ...
```

3 Je otázkou, jestli je nutné zavádět nový operátor. Zvláště pokud někde je nutný a jinde není. Například přiřazení do vstupních proměnných typu reference při volání funkčního bloku vyžaduje `:=` operátor. Podobně i rozhraní, která jsou taktéž reference, nevyužívají tento speciální operátor.

```
END_FUNCTION_BLOCK

PROGRAM prgMotor
  VAR
    Motor    : fbMotorOOP;
    PoruchaGlob : bool;
  END_VAR

  // samostatné přiřazení před voláním bloku
  Motor.refPoruchaGlob ref= PoruchaGlob;
  Motor();

  // přiřazení při volání bloku
  Motor(refPoruchaGlob := PoruchaGlob);
END_PROGRAM
```

U funkčního bloku, který má definované nějaké vstupně-výstupní proměnné, je nutné je přiřadit při jeho volání. U referencí již přiřazení hlídáno není. Pokud není reference přiřazena, tak podobně jako u použití nepřiřazeného rozhraní dojde k chybě za běhu aplikace. CoDeSys nabízí speciální operátor `__ISVALIDREF` pro ověření naplnění reference. Bohužel tento operátor nelze použít pro test rozhraní.

Některé vyšší jazyky (např. C#) automaticky ke všem objektům přistupují přes referenci, přímý přístup používají pouze pro práci s proměnnými základních datových typů. Díky tomuto jednoduchému pravidlu nemusí řešit různé operátory přiřazení.

6 ZÁVĚR

Hlavním přínosem rozšíření normy o objektově orientované programování je větší možnost znovupoužití kódu díky dědičnosti a možnost tvorby obecných vazeb mezi funkčními bloky díky polymorfizmu. To umožňuje vytvářet robustní aplikace, zkrátit dobu potřebnou na jejich tvorbu a následnou údržbu. Díky dělení kódu do metod je možné využít některé z principů událostně řízeného programování, ale pro jeho pohodlnější použití by byla nutná větší podpora na úrovni jazyka. Zůstává zachován původní přístup, který lze kombinovat z novými prvky.

Nové koncepty programování dodávají další rozměry a možnosti při tvorbě aplikací. Do popředí se začíná dostávat vlastní softwarová architektura a s ní související obory (např. návrhové vzory), které do této doby byly v oblasti programování průmyslových aplikací neznámou věcí. Na druhou stranu to ale bude vyžadovat vyšší nároky na tvůrce aplikace.

Otázkou je do jaké míry se navrhované rozšíření podaří prosadit do finální podoby nové verze normy a co z toho jednotlivý výrobci řídící techniky budou schopni a ochotni implementovat. Již v současné době je u většiny výrobců implementace této normy různá. Často nejsou implementovány všechny její prvky nebo jsou implementovány odlišně od normy. Občas se vyskytují vlastní rozšíření, která nejsou s normou kompatibilní. Implementace nové verze normy bude jistě znamenat další investice do vývoje.

Koncept objektově orientovaného programování ve vyšších jazycích jde ještě dále. Nabízí např. vícenásobnou dědičnost, přetěžování, dynamickou alokaci paměti, rekurzi, statické metody nebo zmiňovanou podporu událostně řízeného programování. V souvislosti s tím pak vyvstává otázka, jestli a kdy se přejde na programování řídících automatů v některém z vyšších jazyků a co tomu v současné době ještě brání.

7 SEZNAM POUŽITÉ LITERATURY

[1] Nagel, CH., Evjen, B., Glynn, J., Watson, K., Skinner, M., Jones, A.: Professional C# 2005, Wiley Publishing, Inc., 2006

[2] Reynhold-Heartle, R. A.: OOP with Microsoft Visual Basic .NET and Visual C# .NET Step by Step, Microsoft Corporation, 2002

[3] Mezinárodní standard IEC 61131-3, druhá verze

[4] Nápověda vývojového prostředí CoDeSys

[5] Nápověda vývojového prostředí Mosaic

[6] John, K., Tiegelkapmp, M.: IEC 61131-3: Programming Industrial Automation Systems, Secound Edition, Springer, 2010

[7] Wener, B.: Object-oriented Extension for IEC 61131-3, IEEE Industrial Electronic Magazine, Decenber 2009

[8] Future Extensions of the IEC 61131-3, ETFA 2008, 15th September 2008

[9] Ferg S.: Event-Driven Programming: Introduction, Tutorial, History, 2006

[10] Wikipedia [online]. Dostupné na URL: <www.wikipedia.com>

8 SEZNAM ZKRATEK

CFC	jazyk volných funkčních bloků (z ang. Continuous Function Cart)
ČSN	česká technická norma
FBD	jazyk funkčních bloků (z ang. Function Block Diagram)
GUI	grafické uživatelské rozhranní (z ang. Grafic User Interface)
IEC	International Electrotechnical Commission
IL	instrukční jazyk (z ang. Instruction List)
LD	jazyk reléových schémat (z ang. Ladder Diagram)
OOP	objektově orientované programování
PLC	programovatelný logický automat (z ang. Progammable Logic Controller)
POU	programová organizační jednotka (z ang. Program Organisation Unit)
SFC	sekvenční řízení (z ang. Sequential Function Chart)
ST	jazyk strukturovaného textu (z ang. Structured Text)