VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DISERTAČNÍ PRÁCE

k získání akademického titulu Doktor (Ph.D.)

ve studijním oboru
INFORMAČNÍ TECHNOLOGIE

Ing. Pavel Cagaš

NÁVRH PROGRAMOVACÍHO JAZYKA PRO SYSTÉM RYCHLÉHO VÝVOJE ŘÍDICÍCH A VIZUALIZAČNÍCH APLIKACÍ REÁLNÉHO ČASU

Školitel: Prof. Ing. Jan Honzík, CSc
Datum státní doktorské zkoušky: 15. června 2005
Datum odevzdání práce: 2. srpna 2006
Práce je k dispozici v knihovně Fakulty informačních technologií VUT v Brně

ABSTRAKT

Obsahem práce je návrh jednoduchého programovacího jazyka, jeho překladače a virtuálního stroje, specializovaného na průmyslové aplikace reálného času. Data zpracovávaná algoritmy těchto aplikací zpravidla nejsou okamžitě k dispozici. Původ dat je v řízené technologii – v průmyslových automatech, vstupně/výstupních jednotkách a případně v jiných počítačích zapojených do lokální sítě, přičemž zpoždění způsobené komunikací převyšuje dobu potřebnou ke zpracování informace často o mnoho řádů. Práce navrhuje rozšíření programovacího jazyka o prvky usnadňující tvorbu aplikací reálného času, zavádí mechanismy optimalizace asynchronní komunikace s technologickými jednotkami a také navrhuje implementaci virtuálního stroje zohledňujícího tyto optimalizace. Závěr práce ukazuje praktickou implementaci jazyka OCL a jeho virtuálního stroje se všemi výše popsanými vlastnostmi a jeho skutečné použití v průmyslových, laboratorních i školních podmínkách.

OBSAH

SEZNAM ILUSTRACÍ

SEZNAM TABULEK

1. ÚVOD

1.1. Programovací jazyky a nástroje rychlého vývoje aplikací

Vytvoření programovacího jazyka vysoké úrovně a jeho překladače značně usnadnilo způsob programování počítačů a tím umožnilo jejich využití podstatně širší skupině uživatelů. Počítače, jakožto univerzální stroje na zpracování informací, jsou ale natolik všudypřítomné a možnosti jejich využití natolik rozsáhlé, že znalost jejich programování (zápisu algoritmu v programovacím jazyce) je mezi typickými uživateli spíše výjimečná. Velké množství hotových aplikací specializovaných na konkrétní úkol nabízí uživatelům řešení jejich problémů a propracovaná uživatelská rozhraní jim v ideálním případě dávají zapomenout, že pracují s počítačem.

Hotové aplikace ale ne vždy pokrývají všechny potřeby uživatelů. Často je nutné provádět operace, které přesahují funkčnost zabudovanou v použitých programech. Proto jsou aplikace doplňovány možnostmi programového řízení (často nazývaného „skriptování“ z anglického script – scénář). Zabudovaný programovací jazyk zpravidla zpřístupňuje všechny funkce dané aplikace a umožňuje je rozšiřovat o vlastní algoritmy. Mohou jej využít buď velmi pokročilí uživatelé znalí programování nebo jej použije vývojový tým např. k přizpůsobení standardní kancelářské aplikace specifickým firemním potřebám.

> *Typickým zástupcem programovacích jazyků rozšiřující možnosti aplikací je Microsoft VBA (Visual Basic for Applications). Tento nástroj je základní programovací prostředek balíku kancelářských aplikací Microsoft Office a protože je nabízen i jako vývojové prostředí pro další firmy, lze jej nalézt i mimo sadu Office v řadě jiných aplikací.*

Zcela samostatnou kategorii tvoří nástroje pro grafický vývoj aplikací, nevyžadující od tvůrců znalosti programování. V některých oborech, jako například tvorba řídicích sekvencí průmyslových automatů, laboratorní testování a řízení experimentů, výuka na technických školách apod., je role těchto nástrojů nezastupitelná. V pracovním prostředí vývojového nástroje je možné sestavování „algoritmu“ symbolickou cestou. Předpřipravené komponenty, reprezentující např. čtení či zápis hodnot, logické a aritmetické operace apod., jsou myší přetahovány na pracovní plochu a datové toky jsou reprezentovány grafickými spojnicemi. K tvorbě takových aplikací není zapotřebí zvládnutí programování v obecně chápaném významu – tvůrci aplikací se obejdou bez znalostí syntaxe nějakého jazyka, bez nutnosti definovat datové typy, deklarovat proměnné,

zapisovat podmínky, cykly apod. I když možnosti přizpůsobení grafických nástrojů specifickým potřebám různých nasazení jsou větší než u specializovaných aplikací, i zde je velmi snadné narazit na limity daného nástroje. Schopnost rozšířit funkčnost pomocí nějakého programovacího jazyka možnosti těchto nástrojů značně zvyšují.

1.1.1. Striktně a volně typované (dynamické) jazyky

Mnohé skriptovací programovací jazyky se odlišují od „tradičních“ programovacích jazyků (např. Pascal) především naprostou typovou volností (loose-typed languages). Typovou volností není míněna jen např. rozšířená typová kompatibilita mezi numerickými typy (jako např. v jazyce C), ale spíše naprostá absence typové kontroly. Takové jazyky bývají nazývány dynamické jazyky nebo volně typované jazyky (např. Python). Uvažme programový fragment:

```
…
v = "řetězec"
…
v = 2.7818
…
```

Takto zapsaný program by ve striktně typovaném (nebo též staticky typovaném) jazyce nebylo možné přeložit. Každá proměnná vyžaduje jednoznačné určení typu před prvním použitím (při deklaraci) a změna typu v rámci životnosti proměnné není přípustná.

Otázkou je, jak takový zápis přeložit. Možností je mnoho, jednou z nich je pracovat s každou proměnnou jako s objektem (což samozřejmě předpokládá, že mechanismy objektově orientovaných vývojových nástrojů – zapouzdření, dědičnost, polymorfismus – jsou k dispozici). Každá proměnná může být reprezentována instancí třídy s variantním datovým obsahem a s řadou přetížených operátorů přiřazení pro všechny jednoduché typy v jazyce se vyskytující a pro typ reprezentující objekt. Pokud jsou veškeré složené typy jazyka také implementovány objekty, je množina přetížených operátorů úplná, přiřazení pracuje se všemi typy.

Z výše uvedeného je zřejmé, že typová volnost komplikuje překlad kódu a negativně ovlivňuje jeho výslednou efektivitu (reprezentovat každou proměnnou instancí objektu je jistě náročnější na paměťový prostor i čas než např. reprezentace čísla v jednom slově paměti). Přináší ale další důsledky, např. v případě volání metod objektům. Uvažme jednoduché volání:

```
…
object.method()
…
```

Překlad tohoto volání metody instance objektu (odkazovaného proměnnou *object*) vyžaduje v případě pevně typovaných jazyků znalost třídy instance na niž proměnná *object* odkazuje. Třída musí být explicitně definována a v rámci životnosti proměnné není možné typ změnit. Překladač tak může vyhledat metodu *method* v popisu dané třídy a prověřit korektnost celého příkazu. Stejně při generování kódu může překladač vygenerovat instrukce volání dané metody.

V případě dynamického jazyka ale není v době překladu třída daného objektu známa. Volnost v přiřazení libovolných typů umožní programátorovi naplnit proměnnou *object* odkazem na libovolnou instanci libovolné třídy. Překlad takového volání se tím silně komplikuje.

Proto základní pravidlo překladačů dynamických jazyků zní: odložit vytvoření vazby na metodu na nejpozdější možný okamžik, což v praxi znamená až na okamžik volání. To vyžaduje rozsáhlou runtime podporu jazyka samotného – překladač na místo kódu volání metody generuje kód volání pomocné funkce realizující pozdní vazbu. Implementace takové pomocné funkce samozřejmě předpokládá dostupnost typové informace za běhu programu. Výše zmíněné volání tak může být přeloženo např.:

```
object.GetType().GetMethod("method").Invoke( object )
```

Implementace pozdní vazby silně ovlivňuje zpracování chyb. Zatímco u striktně typovaného jazyka je již při překladu možno zkontrolovat správnost volání (*object* musí být reference na instanci třídy, třída této instance musí mít metodu *method*, která nemá žádné parametry), u volně typovaného jazyka je nutno všechny tyto kontroly provést v okamžiku volání.

Je zřejmé že ačkoliv pozdní vazba přináší neobyčejnou flexibilitu, způsobuje také nebývalou režii. Existuje řada technik, jak generovaný kód optimalizovat, v každém případě bude vykonávání kódu dynamického jazyka výrazně pomalejší než je tomu o striktně typovaných jazyků. Otázka je, nakolik to může být překážka jejich použití. Vzhledem ke spotřebě času jiných činností (databázové dotazy, komunikace po síti apod.), může být režie dynamických jazyků v kontextu celé aplikace nevýznamná.

Poznamenejme, že „dynamičnost“ jazyka souvisí s jazykem samotným (přesněji s implementací jeho překladače, runtime podporou apod.) než s cílovou platformou. To

je zřejmé, pokud je cílová platforma přímo nativní kód nějakého procesoru, ale platí to i pokud je cílem překladu nějaký virtuální stroj, např. Microsoft CLR (Common Language Runtime) pro .NET Framework. CLR je striktně typované prostředí, přesto pro něj existují překladače volně typovaných jazyků, např. IronPython (Python pro .NET).

Zůstává tedy otázka, zda typová volnost přináší nějaké výhody a pokud ano, nakolik vyvažují komplikace při překladu a režii při běhu. Vhodnost „programovacího stylu" dynamických jazyků je do značné míry subjektivní. Typová volnost zřejmě skutečně urychluje tvorbu velmi jednoduchých skriptů automatizujících rutinní činnosti. Nicméně hlavní příčinou krachu projektů v oboru informačních technologiích jsou lidské chyby. Proto je maximální omezení možností chybovat při vývoji programu jeden z hlavních návrhových cílů nově se objevujících jazyků (Java, C#). Typické je např. omezení volnosti v přístupu k paměti a implementace automatického vracení paměti (garbage collector), hlídání rozsahu polí, vyřazení přetěžování operátorů z definice jazyka apod. Jazyk Java, stejně jako např. C#, jsou striktně typované jazyky a typová volnost není v prostředích Java ani .NET vnímána jako pozitivní vlastnost.

I velice rozšířený jazyk Visual Basic v poslední verzi .NET vyžaduje typovou čistotu. Aby bylo možno pracovat s VB jako s dynamickým jazykem, je vyžadováno explicitní vypnutí typové kontroly „Option Strict Off" na počátku programového kódu.

1.1.2. Komponentový model a možnosti programového řízení

Schopnost spolupráce algoritmů zapsaných v nějakém programovacím jazyce s částmi existující aplikace není samozřejmá ani jednoduše implementovatelná vlastnost. Možnost programového řízení musí mít návrháři aplikace od počátku na mysli a musí strukturu aplikace navrhnout tak, aby uživatelům (programátorům) zpřístupnili klíčové funkce aplikace.

Určitou výjimku tvoří programové nástroje určené pro použití z příkazové řádky, schopné číst ze standardního vstupu a zapisovat na standardní výstup. Pokud operační systém dovoluje standardní vstup a výstup programově přesměrovat, spolupráce skriptovacího jazyka s takovými programy není příliš komplikovaná. Ačkoliv se tento koncept může jevit jako jednoduchý a účinný, je vhodný spíše jen pro neinteraktivní dávkové aplikace a pro událostmi řízené aplikace s grafickým rozhraním není k užitku. Také pokud rychlost zpracování hraje roli (např. u vytížených serverových aplikací),

režie spouštění procesů a spolupráce prostřednictvím diskových souborů (byť umístěných ve vyrovnávací paměti souborového systému) přináší významné zpomalení.

Obecným řešením problému řízení aplikací nebo jejich částí programově je komponentový návrh. Vlastnosti požadované po programové komponentě jsou podmnožinu vlastností charakterizujících instanci objektu v objektově-orientovaných vývojových prostředích. Na rozdíl od instance objektu není součástí definice komponenty její stavová informace, datový obsah je zcela skryt (stav komponenty není zvenku přístupný, pokud jej sama komponenta nezpřístupní). Obecně také není definována žádná hierarchie dědičnosti. Jediné co komponenty charakterizuje je schopnost vytvářet instance (např. prostřednictvím určité metakomponenty) a její programové rozhraní – seznam metod (procedur či funkcí) včetně typů jejich parametrů, volacích konvencí, návratových hodnot apod. Samozřejmě je možné implementovat komponenty s použitím objektově-orientovaných vývojových nástrojů, na rozdíl od těchto nástrojů ale komponentový systém nebývá svázán s určitým specifickým programovacím jazykem či vývojovým nástrojem. Rozhraní komponent je ale zpravidla binární a daný programovací jazyk musí mít vlastnosti umožňující tento binární standard dodržet.

Příkladem může být standard volání metod rozhraní komponentového systému COM (Component Object Model). Rozhraní je vektor ukazatelů na funkce a jeho binární podoba odpovídá tabulce virtuálních metod tvořené překladačem jazyka C++. Každá instance C++ třídy s virtuálními metodami odpovídajícími signatuře metod rozhraní COM objektu je tedy implementací tohoto rozhraní.

Pokud je zapotřebí volat funkci či metodu objektu z nějaké knihovny v prostředí „tradičních“ programovacích jazyků, je zapotřebí poskytnout překladači typovou informaci o volané funkci či metodě. Typová informace bývá obsažena např. v textovém hlavičkovém souboru (např. soubory s příponou .h u jazyků C/C++) či binární předkompilované struktuře (tzv. unit u jazyků Pascal/Delphi). Každopádně u tradičních překladačů tato typová informace musí být k dispozici v okamžiku překladu. Tato podmínka je zpravidla u skriptovacích jazyků nesplnitelná – uživatel hodlá volat metody komponent aniž by měl k dispozici např. hlavičkové soubory.

Bez typové informace lze bezpečně volat metody u dynamických (volně typovaných) jazyků. Volání je za běhu zabaleno do pomocných funkcí a data (parametry) do obecných variantních struktur. Volaná metoda nepředpokládá určité typy a typová nekompatibilita neohrožuje stabilitu celého programu. S voláním je ale spojena značná režie, která nemusí

být na závadu, pokud probíhá jen několik volání např. v návaznosti na akce uživatele. U náročných a vytížených aplikací ale může nárůst režie volání až o několik řádů zabránit správné činnosti aplikace. U striktně typovaných jazyků je režie výrazně menší, avšak typová informace je nezbytná. Z těchto důvodů je u komponentových systémů k dispozici typová informace dostupná za běhu programu.

Způsob získávání typové informace je u jednotlivých systémů velmi rozdílný. Např. komponenty systému COM registrují v registrační databázi operačního systému tzv. typové knihovny – binární soubory s typovými informacemi. Systém pak obsahuje API funkce (přesněji nabízí třídy s potřebnými rozhraními) pro vyhledávání v typových knihovnách. Oproti tomu typová informace Java appletů a aplikací je obsažena přímo v kódu v tzv. Constant Pools.

1.2. Reaktivní a synchronní jazyky

Pohled na programovací jazyky a programové systémy s jejich pomocí implementované ze zcela odlišného úhlu reprezentuje následující dělení [16]:

- *Transformační systémy* počítají výstupní hodnoty po zadání vstupních hodnot. Po dokončení výpočtu se zastaví. Transformační systémy bývají například numerické výpočetní programy a simulace.
- *Interaktivní systémy* neustále interagují se svým okolím. Počítač (interaktivní systém) může být považován za řídicí prvek interakce – uživatel žádá systém o službu, ale systém na žádost reaguje jen když momentálně může reagovat a službu poskytne, jakmile dokončí potřebné operace. Typickými příklady interaktivních systémů jsou rozhraní operačních systémů, databázové systémy, distribuované služby poskytované v prostředí Internetu (WWW) apod.
- *Reaktivní systémy* (také nazývané *reflexní systémy*) kontinuálně reagují na podněty okolí a do okolí posílají své podněty. Reaktivní systémy jsou tedy řízeny svými vstupy a musí reagovat v tempu udávaném svým okolím. Řídicím prvkem tedy není systém sám, ale okolní prostředí. Systémy řízení procesů nebo signálové procesory jsou typickými příklady reaktivních systémů.

Rozsáhlé informační systémy málokdy spadají do jediné kategorie. Jednotlivé části odpovídají různým kategoriím podle svého určení. Část systému řídicí technologii je spíše

reaktivní systém, část systému poskytující rozhraní uživatelům nebo obsluhující firemní databázi je spíše interaktivní systém.

Pro lepší ilustraci podstaty interaktivních a reaktivních paralelních systémů bývají používány analogie s reálným světem [15]:

- *Chemický model* představuje systém jako molekuly v roztoku, jejichž polohu a rychlost definuje Brownův pohyb. Komunikace a výpočty jsou prováděny jakmile se dvě nebo více molekul potkají. Interakce může vést k zániku některých starých molekul a případně ke vzniku nových molekul.
- *Newtonovský model* představuje komponenty výpočetního systému jako planety pohybující se na dráze kolem Slunce. V každém okamžiku se každá planeta nalézá v přesně daném bodu prostoru a pohubuje se přesně danou rychlostí. Další pohyb planety je definován její vlastní setrvačností a také přitažlivostí (tedy vzdáleností a hmotností) jiných planet. Prvky výpočetního systému jsou tedy jako planety komunikující svou polohu, rychlost a hmotnost se všemi jinými prvky s nulovým dopravním zpožděním.
- *Vibrační model* rovněž užívá analogiemi s molekulami či atomy, tentokrát ale organizovanými do krystalové mřížky. Jakmile jedna molekula obdrží impuls, předá jej dále sousedním molekulám a impuls se šíří krystalem danou rychlostí (např. rychlostí zvuku).

Chemický model představuje nedeterministický a asynchronní systém. Nelze garantovat, že se dvě molekuly setkají, aby mohly interagovat. Tento model může být popisem interaktivních systémů – nelze předpokládat, kdy a jestli vůbec uživatel spustí danou akci.

Newtonovský model je perfektně deterministický a synchronní. Vystihuje podstatu synchronních jazyků, kde se rovněž předpokládá, že procesy si deterministickým způsobem a okamžitě (bez dopravního zpoždění) vyměňují informace.

Při implementaci řídicích systémů ale často musíme použít složitější vibrační model, kde se informace šíří definovanou rychlostí, kde hrají roli vzdálenosti a kde existuje určitý interní indeterminismus, který bychom ale měli mít pod kontrolou.

Rozpor mezi přesností a přiměřeností modelu je dobře znám ve fyzice. Pro pohyb planet lze použít Newtonovskou gravitaci, která popisuje přitažlivou sílu jako nekonečně rychle se šířící interakci a absolutním čase a prostoru. Také lze ale použít Einsteinovu obecnou teorii

relativity, která popisuje gravitaci jako zakřivení prostoročasu šířící se konečnou rychlostí světla. Nutno poznamenat, že použití Newtonovy teorie je pro výpočty nesrovnatelně snadnější ve srovnání s obecnou teorií relativity. Přitom odchylky nebudou pro planety naší Sluneční soustavy prakticky pozorovatelné (s výjimkou planety Merkur, která obíhá nejblíže ke Slunci a jeho dráha je nejvíce ovlivňována jeho přitažlivostí).

Stejně tomu může být i při implementaci řídicích systémů. Výrazné zjednodušení implementace může být přesto vhodné pro řešení naprosté většiny úloh. Typickým příkladem může být simulace asynchronních elektrických obvodů. Výsledná logická funkce je realizována sledem hradel. Každé hradlo zavádí do šíření signálu jisté zpoždění. I vodivý spoj mezi hradly dané délky představuje zpoždění v šíření signálu. Pokud jsou výstupy vyhodnocovány s určitou periodou, která zaručí rozšíření vstupních signálů logickou sítí, můžeme se na celý systém dívat newtonovsky a řešení celého problému se velmi zjednoduší. Pokud ale bude časování velmi rychlé, musíme počítat se zpožděními na hradlech a model se značně zkomplikuje – systém pak odpovídá spíše vibračnímu modelu.

1.2.1. Synchronní programové systémy

Programovou analogií synchronních elektrických obvodů jsou systémy založené na programových smyčkách. Tento model je u programového řízení procesů velmi častý. Implementace periodicky opakuje tři kroky:

1. čtení vstupů
2. výpočet reakce
3. zápis výstupů

Pokud se vstupní hodnoty změní během výpočtu, jsou brány do úvahy až při dalším průchodu smyčkou. To činí reakci systému atomickou a deterministickou.

Pokud můžeme dobu průchodu programu smyčkou vzhledem k okolnímu prostředí zanedbat, můžeme na systém nahlížet jako na newtonovský (např. odkalovací nádrž čistírny odpadních vod dozajista nepřeteče během zlomků sekund). Jestliže je nutno brát dobu provádění smyčky do úvahy, odpovídá tento systém vibračnímu modelu (např. zastavení motoru dopravníku po sepnutí koncového spínače může vyžadovat reakci v jednotkách milisekund, náročná programová smyčka může být prováděna i o několik řádů déle).

I mezi synchronními jazyky lze rozlišit dvě hlavní větve v závislosti na použitém programovém stylu:

- Synchronní jazyky řízené tokem dat, vhodné zejména pro zpracování souvislých datových proudů. Typickými zástupci této třídy jsou jazyky Lustre a Signal.
- Imperativní synchronní jazyky vhodné pro implementaci řídicích algoritmů. Typickým zástupcem těchto jazyků je jazyk Esterel.

1.2.2. Synchronní jazyky řízené tokem dat – jazyk Lustre

Řízení tokem dat se uplatňuje převážně u řízení ustálených procesů a u zpracování datových toků signálovými procesory. K odvození vstupů jsou typicky používány okamžité hodnoty vstupů a hodnoty z předešlého kroku (z předešlého průchodu řídicí smyčkou).

Typickým zástupcem synchronních reaktivních jazyků řízených tokem dat je jazyk Lustre [12] [13]. Základní výpočetní jednotkou jazyka Lustre je podprogram zvaný „uzel“ (node). Lustre operuje s daty v podobě „proudů“ (streams). Proud je sekvence hodnot. Všechny hodnoty daného proudu jsou stejného typu. Každý program v jazyce Lustre má podoby cyklu (smyčky), přičemž každému průchodu smyčkou odpovídá jedna pozice ve vstupních a výstupních proudech – v n-tém průchodu smyčkou se pracuje vždy s n-tým prvkem každého proudu. Každý uzel má jeden nebo více vstupních parametrů a jeden nebo více výstupních parametrů.

Jako příklad použití jazyka Lustre uveďme uzel indikující výskyt vzestupných hran v proudu logických hodnot. Za vzestupnou hranu budeme považovat situaci, kdy ve se vstupním proudu logických hodnot X objeví logická jednička následně po logické nule (x_n = false, x_{n+1} = true). Tento stav budeme indikovat ve výstupním proudu Y hodnotou y_{n+1} = true, následující hodnoty v Y budou mít opět hodnotu false. Zápis v jazyce Lustre bude vypadat následovně:

```
node Edge (X : bool) returns (Y: bool);
let
  Y = X and not pre( X );
tel
```

Jazyk Lustre pracuje zcela intuitivně s operátory (+, -, and, not, …), přiřazeními (=) apod. Zvláštností je operátor „pre“, který zpřístupňuje hodnotu daného proudu z předešlého kroku. Poznamenejme, že součástí jazyka je operátor zpřístupňující právě jednu předešlou hodnotu. Pokud algoritmus vyžaduje práci s delší historií, je nutno její paměť implementovat uvnitř daného uzlu např. s použitím polí.

Konstantní hodnoty jsou v jazyce Lustre implementovány rovněž proudy, jejichž všechny prvky mají stejnou hodnotu. Tak zápis „false“ představuje proud logických konstant false, false, …

Určitým problémem je hodnota prvního prvku výstupního proudu Y. Protože hodnota pre(X) pro první prvek není definována (má hodnotu „nil“), není definována ani hodnota prvního prvku Y. Pro ošetření těchto stavů definuje jazyk Lustre operátor „následován“ (zapisovaný ->), který umožňuje proudy inicializovat. Pokud je A proud hodnot a_1, a_2, a_3, … a B proud hodnot b_1, b_2, b_3, …, výsledkem operátoru A -> B je proud a_1, b_2, b_3, … Můžeme tedy uzel Edge z předchozího příkladu zapsat korektně:

```
node Edge (X : bool) returns (Y: bool);
let
  Y = false -> X and not pre( X );
tel
```

První hodnota bude vždy false a následující hodnoty budou již ukazovat vzestupné hrany vstupního proudu.

V rámci implementace každého uzlu lze používat již implementované uzly. Uzel signalizujíc sestupnou hranu může vypadat např. následovně:

```
node FallingEdge (X : bool) returns (Y: bool);
let
  Y = Edge( not X );
tel
```

Součástí výrazu v jazyce Lustre může být i konstrukce if … then … else. Uzel přepínače může být implementován například následovně:

```
node Switch (set, reset, initial : bool) returns (level: bool);
let
  level = if set then true
          else if reset then false
          else pre( level );
tel
```

Jazyk Lustre dovoluje definovat uzly s libovolným počtem vstupních i výstupních proudů. Pro manipulaci s více proudy současně dovoluje syntaxe jazyka slučovat proudy do uspořádaných n-tic (tuples). Například uzel řadící uspořádanou dvojici podle velikosti může vypadat následovně:

```
node MinMax (x, y : int) returns (min, max: int);
let
  (min, max) = if (x < y) then (x, y) else (y, x);
tel
```

Operátor „when“ dovoluje zředění (podvzorkování) libovolného proudu. Nechť B je proud typu boolean a X proud typu *t*. Pak výraz „X when B“ je proud typu *t*, který obsahuje jen ty prvky proudu X, u nichž odpovídající prvky proudu B mají hodnotu true. Operátory jazyka Lustre vyžadují, aby na obou stranách byly použity pouze proudy se shodným vzorkováním.

Jazyk Lustre obecně nedefinuje periodu vykonávání svých uzlů. Počet průchodů za jednotku času tedy záleží na překladači/interpreteru jazyka, na použitém hardware apod. K synchronizaci prováděných akcí s reálným časem lze použít podvzorkování when. Předpokládejme že existuje proud typu boolean nazvaný „seconds“. V tomto proudu bude vždy na začátku každé sekundy hodnota true, jinak false. Pokud chceme např. předešlý uzel MinMax aktivovat každou sekundu, zapíšeme volání následovně:

```
MinMax((x, y) when seconds);
```

Poznamenejme, že jazyk Lustre neposkytuje prostředky pro synchronizaci s reálným časem mimo explicitně předaných proudů časových značek, používaných pro podvzorkování operátorem when. Jazyk rovněž nepostihuje skutečnost, že pokud aplikace skutečně řídí reálný proces, zdroje proudů nejsou v počítači a je nutno data získat z vnějších periferií. Přitom rychlost periferií bývá až o mnoho řádů menší ve srovnání s rychlostí procesorů. Více je tato problematika popsána v kapitole „Dostupnost dat v průmyslových systémech“.

1.2.3. Imperativní synchronní jazyky – jazyk Esterel

Uvažme následující řídicí blok se dvěma signálovými vstupy A a B, inicializačním vstupem R a jedním výstupem. Funkci řídicího bloku popíšeme obecným jazykem následovně: zapiš na výstup 0 jakmile obdržíš hodnoty z obou vstupů A a B. Při obdržení vstupu R inicializuj blok do počátečního stavu. Taková specifikace není příliš jednoznačná, budeme předpokládat že výstup není během aktivity signálu R aktivní a že oba vstupy A a B se mohou objevit současně. Pokud bychom chtěli takový řídicí blok formalizovat např. pomocí automatu, zápis nebude úplně snadný a přehledný.

Účelem imperativních synchronních jazyků je tedy formalizovat zápis složitých řídicích algoritmů snadněji, přehledněji a účelněji. Základní princip lze formulovat větou „vše se zapisuje pouze jednou“ (anglicky Write Things Once – WTO).

> *Snaha zapisovat každou konstrukci pouze jednou není specifickou vlastností synchronních jazyků, stojí za obecnými programovými konstrukcemi jako např. smyčka,*

podprogram (procedura a funkce) apod. Stejný princip motivoval vznik knihoven i objektově orientovaného programování.

V jazyku Esterel bychom výše popsaný řídicí blok zapsali následovně:

```
module ABRO;
input A, B, R;
output O;
loop
  [ await A || await B ];
  emit O
each R
end module
```

Poznamenejme, že v zápisu se všechny signály vyskytují mimo deklarace jen jednou.

Během každé odezvy má každý signál jednoznačný status přítomen/nepřítomen. Přítomnost vstupních signálů je dána okolním prostředím, výstupní signály jsou standardně nepřítomné, objeví se pouze po vykonání příkazu „emit“. Instrukce „await A“ čeká na přítomnost signálu A a po jeho objevení se skončí.

Paralelní kombinace dvou příkazů (operátor ||) skončí jakmile jsou ukončeny oba příkazy. Z hlediska sémantiky jazyka Esterel je doba potřebná pro synchronizaci rovna 0 (odpovídá newtonovskému modelu). Naopak sekvenční řazení příkazů „p; q“ předá řízení příkazu q až po ukončení příkazu p. V předešlém příkladu je tedy výstup O nastaven jakmile oba příkazy skončí.

Zápis „loop p each R“ je tzv. preemptivní operátor. Tělo p je vykonáváno a běží do okamžiku, kdy se objeví signál R. V okamžiku objevení se signálu R je vykonávání těla příkazu p ukončeno bez ohledu na jeho okamžitý stav a vykonávání p je spuštěno opět od počátku. Přednost je v příkazu „loop … each“ tzv. silná, protože má vyšší prioritu než vykonávání těla příkazu. Pokud tedy v předchozím příkladu budou přítomny všechny tři signálu A, B a R, výstup O nebude aktivován. Toto chování významově zcela odpovídá příslušnému automatu, jen zápis je mnohem jednodušší.

Synchronní chování zaručuje, že interní implementace z hlediska běhu algoritmu nespotřebovává čas, je okamžitá. Jediné příkazy, které trvají v čase jsou ty, u nichž je to explicitně požadováno, tedy „await“ a „loop … each“. Tato podmínka je nezbytná, aby chování řídicího bloku odpovídalo automatu bez nepředvídatelných změn stavů.

I další příkazy jazyka Esterel reprezentují různé preemptivní struktury, např. „abort … when“, „every … do … end every“ apod.

Esterel rovněž podporuje mechanismus výjimek, obdobný mechanismu implementovaném v jazyce C++ (try … catch) nebo v rozhraní Win32 API (__try … __except). V jazyce Esterel slouží k zachycení výjimek konstrukce „trap … handle … end trap“.

1.3. Programovací jazyk pro průmyslové systémy

Průmyslové prostředí přináší určitá specifika ve srovnání s tradičním nasazením informačních technologií. Mimo kritických projektů, u nichž jsou náklady na vývoj natolik vysoké, že zařazení týmu špičkových odborníků na informační technologie nehraje roli (např. software v dopravních letadlech či vojenských systémech), je naprostá většina aplikací v průmyslovém prostředí vyvíjena odborníky na řízených procesů a technologií. Znalosti programování a obecně orientace v informačních technologiích bývá proto u tvůrců průmyslových aplikací v naprosté většině případů spíše výjimkou než pravidlem. Pokročilé koncepty, jako např. objektově-orientované programování nebo programování konkurenčních prováděcích toků a jejich synchronizace, jsou zpravidla za hranicemi možností návrhářů aplikací.

Průmyslové aplikace vyžadují vysokou míru bezpečnosti a stability. Chyba nebo výpadek řídicího systému přímo ovlivňuje výrobu a způsobuje často velké škody. U kritických aplikací (např. jaderné elektrárny) jsou nároky na spolehlivost zcela zřejmé.

Možná nebezpečí (snadnost vytvoření chybného programu) přinášená dynamickými jazyky je velmi znevýhodňují v kritických nasazeních. Zejména detekce chyb až za běhu programu (např. volání neexistující metody) velice znesnadňuje testování a ladění aplikací, tak důležité pro úspěšné nasazení.

Ačkoliv aplikace pro průmysl nejsou tvořeny programátory, ale spíše procesními inženýry, jejich rozsah bývá často velmi velký – aplikace obsahující stovky operátorských obrazovek a komunikující s desítkami tisíc vstupně/výstupních bodů nejsou výjimkou.

Požadavky zákazníků často přesahují rámec možností grafického vývoje aplikace a vyžadují zápis algoritmu v programovacím jazyce. Tento jazyk by ale měl zohledňovat své specifické použití.

- Programovací jazyk musí být co nejbezpečnější, detekující maximální množství chyb již za překladu. Musí být striktně typový.
- Programovací jazyk musí umožňovat synchronizaci s reálným časem. Musí tedy nést prvky reaktivních systémů. Současně ale je součástí průmyslových aplikací

interaktivní uživatelské rozhraní a jazyk musí podporovat událostmi řízené zpracování.

- Programovací jazyk netvoří jádro celého vývojového systému, ale pouze jeho rovnocennou část spolu s řadou komponent specializovaných na typické problémy, editačních nástrojů apod. Přesto je nutné aby množina komponent podporovala programové řízení, zveřejňovala svá rozhraní apod.

1.3.1. Dostupnost dat v průmyslových aplikacích

Systémy průmyslové automatizace typicky spolupracují se vstupně/výstupními jednotkami, které pro ně zajišťují čtení dat z řízeného procesu a zápis dat do řízeného procesu. I pokud je taková vstupně/výstupní jednotka realizována jako zásuvná karta na lokální sběrnici počítače (např. PCI), je její doba odezvy řádově pomalejší než je rychlost zpracování instrukcí počítače. Pokud je ale realizována jako samostatná jednotka komunikující po sériovém rozhraní (např. RS-485), poměr rychlostí zpracování informace a jejího získání může dosáhnout až řádu $1:10^9$.

Pokud budeme nadále hovořit o programovém vybavení pro systémy průmyslové automatizace, budeme mít na mysli programy pracující na počítačích v prostředí univerzálních operačních systémů. Použití mikrokontrolérů (mikroprocesorů integrovaných spolu s pamětí a periferiemi do jediného čipu) je natolik rozšířené, že programy dnes pracují v řade čidel, převodníků, inteligentních periferiích apod. Tyto programy bývají zpravidla relativně krátké (v řádu stovek až tisíců instrukcí) a úzce specializované. Nejsou na ně kladeny nároky na možnosti skriptování, na vývoj ad-hoc aplikací, na interoperabilitu s jinými systémy, na spojení s firemními databázemi apod. Také spolupráce těchto programů s periferiemi je specifická pro danou platformu.

Komunikace se vzdálenými periferiemi je také nutně potenciálně nespolehlivá a aplikace s tím musejí počítat. Určitá třída chyb je snadno odhalitelná (např. chybný kontrolní součet), pokud ale např. dojde k fyzickému poškození koncového zařízení nebo přerušení komunikační linky, často jediným prostředkem ošetření takové situace je definice maximální časové prodlevy komunikace (timeout). Velké dopravní zpoždění a potenciální chybovost jsou hlavní charakteristiky přenosu dat v průmyslových systémech.

1.3.2. Optimalizace přístupu k datům

Výše zmíněná omezení vyžadují specifický přístup ke komunikacím s průmyslovými periferiemi. Cílem je maximalizovat objem přenesených dat a minimalizovat dopravní zpoždění.

Základní optimalizace spočívá v maximální možné kumulaci komunikačních požadavků pocházejících z různých částí aplikace (různých komponent, různých procedur skriptovacího jazyka apod.) v daném časovém okamžiku. Kumulace zvyšuje propustnost celého systému, protože:

- Přenos dat po komunikačních linkách se neobejde bez režie. Navazování spojení, kontrolní součty či CRC kódy, potvrzování paketů – to vše zmenšuje efektivní přenosovou kapacitu. Součet časů přenosu dvou bloků dat dvěma transakcemi je vždy větší než pokud je stejný objem užitečných dat přenesen v rámci jediné transakce.
- Přenosový protokol bývá přizpůsoben konvencím daného zařízení. Data v průmyslových automatech jsou často sdružována do logických bloků podle typu či účelu. Přenosový protokol pak např. dovede přenést jen celý blok, nikoliv jeho část. Opakující se požadavky na data z různých míst jediného bloku tak mohou vést k opakovanému nadbytečnému přenosu celého bloku.

Dalším způsobem optimalizace je eliminace nepotřebných datových přenosů. Komunikace mezi aplikací a periferiemi probíhají v diskrétních časových krocích, přičemž typicky ne všechna data jsou v daném časovém kroku zapotřebí. Naivní přístup, přenášející periodicky veškerá data použitá v daném algoritmu, může vést k zahlcení komunikačních linek a tím k nutnosti prodloužit komunikační periody se všemi důsledky pro kvalitu řízení a odezvu systému.

Inteligentní návrh aplikace a explicitní vyžádání komunikace jen pokud je to v kontextu aplikace nutné je jistě velice žádoucí, v některých případech ale ani uživatel není schopen provést explicitní optimalizaci bez podpory celého systému. Uvažme situaci, kdy je zapotřebí přenést jen jeden ze dvou bloků dat v závislosti na jiné veličině, rovněž měřené ve stejném časovém kroku. Uživatel v zápisu přenosu dat zřejmě použije podmíněný příkaz „if“, do jehož dvou větví zapíše přenos jednoho a druhého bloku. Dále již nemá možnost komunikaci ovlivnit. Je na překladači a virtuálním stroji programovacího jazyka, aby na

základě vyhodnocené podmínky vyvolal komunikaci pouze žádoucích dat. Návrh takového jazyka a odpovídajícího virtuálního stroje je hlavním cílem této práce.

2. OCL – PROGRAMOVACÍ JAZYK PRO PRŮMYSLOVÉ APLIKACE

Jazyk OCL (Object Control Language) byl navržen a implementován jako programovací jazyk systému rychlého vývoje průmyslových řídicích a vizualizačních aplikací Control Web. Samotný jazyk i jeho virtuální stroj prodělal spolu s celým systémem evoluční vývoj. Specifické použití tohoto jazyka motivovalo zabudování řady unikátních vlastností, které OCL odlišují od jiných skriptovacích jazyků.

> *Syntaxe jazyka vychází řady modulárních jazyků vyvinutých prof. N. Wirthem na ETH Zürych. Zcela základním návrhovým cílem jazyka bylo zachování jednoduchosti a jednoznačnosti zápisu algoritmu s minimální množinou základních konstrukcí (řídicích příkazů), což je pro jazyky vytvořené prof. Wirthem typické.*

2.1. OCL v systému Control Web

Ačkoliv je jazyk OCL integrální součástí systému Control Web, jeho použití není pro návrh aplikace nezbytné. Control Web nabízí, vedle klasického programátorského přístupu, také možnost vytvořit aplikaci zcela bez programování. Podstata tvorby aplikace bez programování spočívá v zabudovávání (vytváření instancí) předpřipravených komponent do aplikace a jejich parametrizace.

Zápis parametrů komponent nepodléhá téměř žádným syntaktickým pravidlům. Za jejich čtení ze zdrojového textu jsou zodpovědné komponenty samotné, překladač provádí pouze lexikální analýzu a poskytuje komponentám proud lexikálních symbolů. Dodržování jistých syntaktických konvencí pro vyjádření podobné sémantiky je věcí kázně návrhářů komponent (a případně administrativních zásahů vedoucího projektu) a není překladačem vynucováno. Tato volnost je dána neobyčejnou rozmanitostí komponent – komponenta může být nejen prosté zobrazení technologické veličiny nebo ovládací prvek grafického uživatelského rozhraní aplikace, ale třeba HTTP server, rozhraní SQL databáze, zobrazení technologického schématu či komplexní třírozměrná scéna představující model technologie.

Kontroly překladače jsou jen minimální, např. je zabráněno komponentě pokračovat ve čtení vstupní pásky lexikálních symbolů za hranicí jejího zdrojového textu. Zdrojový text komponenty je ukončen klíčovým slovem složeným z „end_“ a jména třídy komponenty.

Takto vytvořené klíčové slovo tedy není možné použít uvnitř zdrojového textu komponenty.

Komponenty v rámci aplikace nutně komunikují se vstupně/výstupními jednotkami i mezi sebou, odkazují se na globální data apod. Tyto odkazy na data a na jiné komponenty tvoří „datovou“ část rozhraní každé komponenty (další část je programové rozhraní). Pravidla pro zápis parametrů (vlastností) tvořících rozhraní komponenty už ale musí existovat a komponenty k načtení těchto parametrů využívají API systému.

Datové rozhraní komponent je vázáno nejen syntakticky (syntaktická analýza je prováděna API funkcemi systému, volanými k načtení parametrů obsahujících datové elementy), ale také implementačně. Není možné, aby komponenta vytvořila vlastní instance datových elementů mimo struktury spravované jádrem systému. U takto vytvořených datových elementů by nepracovaly základní mechanismy (např. komunikace).

Ačkoliv je parametrizace komponent navrhována pro zpřístupnění tvorby aplikací co nejširšímu okruhu ne-programátorů, přesto se ale uživatelé neobejdou bez zvládnutí dvou základních pojmů:

- datový element
- výraz

2.1.1. Datové elementy – typy a druhy

Ve všeobecně rozšířených programovacích jazycích není pojem „datový element“ zaváděn, neboť o konstantách a proměnných se zpravidla hovoří v různých kontextech. Pokud je to třeba, není problém vyjmenovat obě kategorie. V systému Control Web je ale množina druhů datových elementů rozšířena. Datový element je tedy zobecněný název pro konstanty, proměnné, kanály (jejichž prostřednictvím aplikace komunikuje s vnějším světem), plánované datové elementy (elementy s asociovanými výrazy, periodicky vyčíslovanými plánovačem) apod. S každým datovým elementem je spjat jeho datový typ a také jeho druh. Definice datového typu odpovídá obecně rozšířeným zvyklostem rozlišování typu dat, jež mohou být v datovém elementu uchovávána (číselný typ, logický typ, řetězcový typ). Atribut datového druhu je ale specifický pro Control Web a rozlišuje způsob, kterým se s datovým elementem pracuje. Např. datový element druhu proměnná vyžaduje jiný způsob zpracování než element druhu kanál, jež představuje „datový tunel“

mezi aplikací a vnějším světem (kanály jsou prostřednictvím ovladačů napojeny na vstupně/výstupní jednotky).

Datové elementy uchovávají stav aplikace a bez jejich definice se prakticky žádná aplikace neobejde. Každý uživatel se tedy musí seznámit s dostupnými datovými typy a jejich vlastnostmi a rovněž z datovými druhy (nebo alespoň s jejich podmnožinou, potřebnou pro danou aplikaci). Rozličné druhy datových elementů ovlivňují nejen syntaxi zápisu (např. za běhu nelze přiřadit novou hodnotu do konstanty), ale především sémantiku programu.

Např. syntakticky shodný zápis přiřazení se za běhu bude chovat jinak, pokud na pravé straně bude jen proměnná (její hodnota bude vyčíslena a přiřazena) nebo kanál (kanál bude dopředu „označen“ pro čtení prostřednictvím API ovladače, jádro poté požádá ovladač o komunikaci všech označených kanálů požadovaných v daném časovém kroku, vyčká do dokončení komunikace nebo do vypršení časového limitu a pokračuje s novou hodnotou kanálu, případě s naposledy platnou, pokud časový limit vypršel).

Hlavní motivací rozšíření druhů dat nad rámec obvyklých konstant a proměnných je samozřejmě potřeba komunikovat s technologickými zařízeními (jsou zapotřebí kanály) a také další specifika průmyslových aplikací. Běžné například je, že hodnota přečtená ze vstupně/výstupní jednotky reprezentující např. teplotu je přímo vracena analogově/digitálním převodníkem v jeho vlastním rozsahu a vyžaduje přepočet, aby odpovídala patřičným fyzikálním jednotkám, případně musí být linearizována polynomem aby se kompenzovala nelineární charakteristika teplotního čidla apod. Tyto přepočty jsou vyjádřeny výrazem a pokud původní hodnota přečtená z čidla není v aplikaci zapotřebí, použije se jen hodnota přepočtená. To je umožněno spojením datového elementu s výrazem, který je vyčíslen vždy když je hodnota datového elementu vyžadována kdekoliv v aplikaci.

Pokud by k takovému vyčíslení docházelo skutečně při každém použití, byla by tato konstrukce velice neefektivní. Nicméně Control Web je orientován na aplikace reálného času a definuje časové kroky aktivace virtuálních přístrojů, jež jsou násobkem elementárních systémových časových kroků. K vyčíslení výrazů spojených s datovými elementy dochází tedy vždy jen jednou v každém systémovém časovém kroku. Pokud v daném kroku již byl výraz daného elementu vyčíslen, je použita výsledná hodnota. Control Web nabízí také odlišnou sémantiku datových elementů spojených s výrazem, kdy jsou výrazy vyčíslovány buď periodicky plánovačem nebo explicitně např. v závislosti na komunikaci apod.

Uvažme příklad čtení teploty z analogově/digitálního převodníku. Vztah mezi hodnotou převodníku a skutečnou teplotou je určen dvěma konstantami – posunem a měřítkem:

```
const
  k = 0.0657;
  q = -68.04;
end_const;

channel ain1 {driver = dl; direction = input};
  raw_temperature : real {driver_index = 100};
end_channel;

expression
  temperature = k * raw_temperature + q;
end_expression;
```

Hodnoty „raw_temperature“ představuje výstup A/D převodníku, hodnota „temperature“ pak teplotu ve stupních Celsia. Pokud je hodnota „temperature“ použita v daném systémovém časovém kroku poprvé, je vyčíslen výraz s touto hodnotou spojený. Tento výraz ale odkazuje na datový element druhu kanál. Pokud je v daném systémovém časovém kroku tento kanál už přečten (nějaký jiný výraz obsahoval odkaz na „raw_temperature“), je jeho hodnota použita. Pokud ne, je hodnota změřena ze vstupně/výstupního zařízení. Všechny další odkazy na „raw_temperature“ i na „temperature“ v daném časovém kroku již budou uspokojeny vrácením známé hodnoty.

Pokud by hodnota „raw_temperature“ nebyla v aplikaci využita (což je velmi pravděpodobné), dovoluje syntaxe systému Control Web sekci definice kanálů zcela vynechat – hodnotu kanálu lze získat ze jména ovladače (v našem případě „dl“) a indexu kanálu:

```
expression
  temperature = k * dl.100 + q;
end_expression;
```

2.1.2. Výrazy

Výraz je předpisem pro výpočet a nositelem vypočtené hodnoty a pravidla jeho zápisu uživatel také musí zvládnout. Aplikaci bez zápisu výrazů taktéž prakticky nelze vytvořit (již předešlá diskuse o datových elementech zahrnovala řadu výrazů). Konstrukce výrazů zcela odpovídá zažitým zvyklostem – výraz se v systému Control Web skládá z:

- Literálů reprezentujících hodnoty. Literál je hodnota zapsaná přímo ve výrazu, např. číslo čtyřicet dva je reprezentováno znaky 42. Literálem je také např. zápis

řetězce 'blok 1'. Logické hodnoty mají jen dva stavy a tak pro ně existují jen dva literály reprezentující pravdu a nepravdu – true a false.

- Datových elementů reprezentujících jednu hodnotu. Při vyčíslení výrazu je na místo každého datového elementu použita hodnota tohoto elementu v okamžiku vyhodnocení.
- Operátorů představující operace s hodnotami ať již literálů nebo datových elementů. Operátorem je například sčítání (operátor +), porovnání na rovnost (operátor =) apod. Operátory mohou být binární (pracující se dvěma operandy) nebo unární (pracující s jedním operandem). Operátory bývají většinou reprezentovány speciálními znaky (+, -, *, /, ...), ale existují i operátory definované textovými identifikátory (and, or, xor, ...).
- Funkcí, které stejně jako operátory představují operace s hodnotami. Na rozdíl od operandů nebývá počet parametrů nijak omezen, běžné jsou funkce bez parametru (např. funkce rand() generující pseudonáhodná čásla) i se čtyřmi a více parametry. Funkce bývají vždy reprezentovány textovým řetězcem a argument(y) jsou uvedeny za tímto řetězcem v závorkách.
- Pomocných symbolů — závorek, které pouze mění způsob vyhodnocování výrazů.

Významově se operátory a funkce vlastně shodují, liší se jen způsobem zápisu — např. na místo operátoru + si lze představit funkci add(x, y), která by vracela součet svých operandů. Bez operátorů by tedy bylo možné se zcela obejít a všechny je nahradit funkcemi, zápis výrazů by ale byl poněkud neobvyklý a zbytečně složitý. Které operace jsou reprezentovány operátory a které funkcemi je spíše otázka zvyklostí než nějakého objektivního kritéria. Proto se operandy používají pro operace u nichž je to zažité (např. základní aritmetické operace) a pro něž existuje také zažitá symbolika. V některých případech ani to není pravda, např. pro operaci „druhá odmocnina“ není v zápisu výrazu použit symbol √, obvyklý v matematické literatuře, ale je zavedena funkce sqrt(x).

Výraz není v principu nijak omezen na určité datové typy — výsledkem výrazu může být číslo, řetězec nebo logická hodnota. Ovšem pravidla pro tvorbu správných výrazů obsahují striktní typová omezení týkající se operandů i funkcí. Každý operátor i funkce přesně určují, s jakými typy elementů pracují a jaký je typ výsledku dané operace, a těmto

podmínkám samozřejmě musí výraz plně odpovídat. Pouze číselné typu jsou považovány za vzájemně slučitelné.

Ačkoliv součástí výrazu může být v systému Control Web volání OCL procedury s návratovou hodnotou, podpora výrazů nabízí řadu zabudovaných funkcí (např. matematické funkce, převodní funkce apod.). Důvodem k zabudování nejčastěji používaných funkcí přímo do syntaxe výrazů je fakt, že procedury se v systému Control Web mohou vyskytovat pouze v rámci konkrétních komponent. Systém tedy obsahuje řadu pomocných virtuálních přístrojů, jež plní funkci podobnou knihovnám v klasických programovacích jazycích (tyto virtuální přístroje jsou v aplikaci vždy přítomny, aniž by uživatel vytvářel jejich instance – např. podpora pro výpočty s datem a časem ve virtuálním přístroji „date“, podpora přístupu k souborovým službám ve virtuálním přístroji „file“ apod.).

Dobře si tedy lze představit např. virtuální přístroj „math“, který by obsahoval goniometrické funkce apod. Pomineme-li poněkud vyšší režii volání procedury virtuálnímu přístroji oproti volání zabudované funkce, především volání procedur virtuálním přístrojům vyžaduje od uživatele seznámení se s mechanismy procedur a jejich volání. Jak již ale bylo řečeno, jedním z návrhových cílů systému Control Web je umožnit tvorbu aplikací bez nutnosti zapisovat algoritmy. Ačkoliv nelze objektivně rozhodnout, které funkce je žádoucí zabudovat přímo do syntaxe výrazů a které je již možno přesunout do knihovních virtuálních přístrojů, určité dělení v rámci systému Control Web existuje.

2.2. Výrazy jako nezávislá vrstva virtuálního stroje

2.2.1. Překlad výrazů pro virtuální stroj

Instrukce provádějící základní aritmetické a logické operace jsou (a vždy byly) nedílnou součástí instrukčního souboru všech počítačových procesorů. Každý počítač dovede sečíst dvě čísla a také rozhodnout, zda je daný registr (případně paměťová oblast) nulový či zda je jeden operand větší než druhý (nebo alespoň detekuje přetečení při odečtení operandů).

Podobně je tomu i v případě dnes nejrozšířenějších virtuálních strojů – Java Virtual Machine (JVM) firmy Sun Microsystems a Common Language Runtime (CLR) firmy Microsoft. Oba virtuální stroje jsou zásobníkově orientované. Např. instrukce sečtení dvou čísel nemá „registrovou“ podobu jako v následujícím příkladě (v příkladu uvažujeme tzv.

tří-registrovou aritmetiku, kdy výsledek není ukládán do registru jednoho z operandů, ale do nového – třetího – registru [3]):

```
ldw       R.a, op1
ldw       R.b, op2
add       R.c, R.a, R.b
```

Oproti registrově orientovanému stroji zásobníkový virtuální stroj definuje aritmetické i logické operace nad vrcholem zásobníku (např. [4]):

```
push      op1
push      op2
add
```

Instrukce aritmetického součtu z vrcholu zásobníku odebere dvě čísla a výsledek operace sečtení uloží zpět na zásobník.

Bez ohledu na podobu jsou instrukce virtuálního stroje využívány (podobně jako instrukce skutečného procesoru) i k vyčíslení výrazů – překladač generující kód pro virtuální stroj generuje patřičné instrukce kdykoliv je zapotřebí vyhodnotit výraz (např. při podmíněných příkazech, při překladu přiřazení atd.).

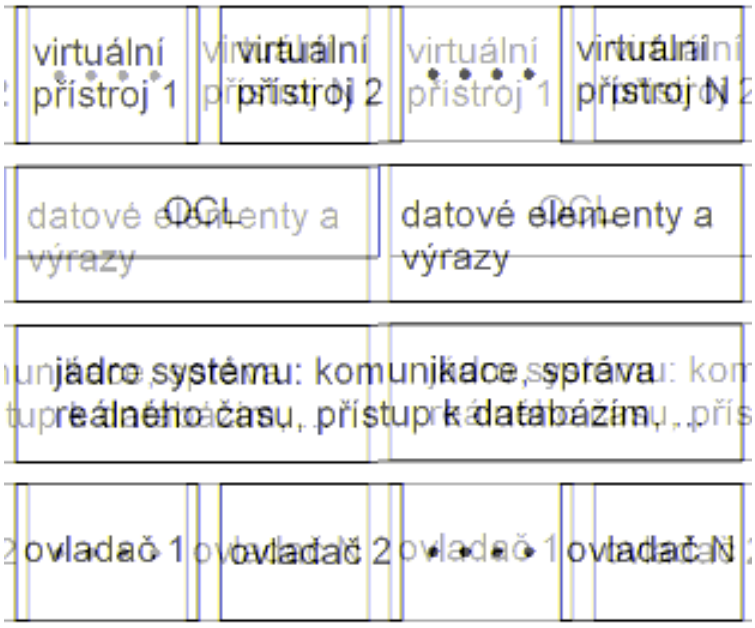
2.2.2. Výrazy v jazyce OCL

Jak bylo zmíněno v obecném nástinu architektury systému Control Web, použití programovacího jazyka je během tvorby aplikace „nepovinné“, použití výrazů se ale prakticky nelze vyhnout (minimálně ve své triviální podobě – i když je hodnota nějaké vlastnosti definována jako jediná konstanta či proměnná, jedná se z hlediska systému o výraz). Výrazy jsou tedy hojně používány pro definici vlastností komponent i pro parametrizaci aplikace samotné.

V principu by bylo možné generovat pro každý výraz část kódu virtuálního stroje a pro jeho vyhodnocení spustit jeho interpretaci. Na místo takové implementace byl ale u systému Control Web použit jiný, původní přístup. Výrazy nejsou překládány do podoby lineárního sledu instrukcí (v podstatě reprezentujícího průchod stromem), ale zůstávají ve stromové datové struktuře. Každý uzel stromu představuje variantní záznam reprezentující operace (operátory a funkce), listy reprezentují operandy (případně reference na operandy).

> *U takto reprezentovaného výrazu se naskýtá otázka paměťové efektivity. Pokud je výraz reprezentován jako sled instrukcí virtuálního stroje, je vždy celý zápis výrazu obsažen v jediné oblasti paměti. Pokud je ale výraz uchován v podobě stromu, představuje každý uzel i list separátně alokovaný blok paměti. S nárůstem počtu bloků rostou nároky na*

správu paměti a překlad se zpomaluje. Vyjdeme-li ale z předpokladu, že každý uzel stromu je variantní záznam a je tedy stejně dlouhý, je možné implementovat vlastní správu paměti pro bloky konstantní délky. Taková správa paměti je velice efektivní – složitost alokování i uvolnění bloku paměti je jednotková, nezáleží na počtu prvků stromu. Přitom z hlediska správy paměti se jedná o alokaci jednoho či více velkých bloků paměti, v jejichž prostoru pracuje vlastní alokace uzlů a listů stromu, a režie s takovou alokací spojená je zanedbatelná.



Obr. 1: Virtuální stroj OCL v architektuře systému Control Web

Mimo překladače, vytvářejícího stromovou strukturu, je zapotřebí interpreter schopný takto reprezentovaný výraz vyčíslit. Při vyčíslení výrazu interpreter rekurzívně prochází stromem, vyčísluje podstromy reprezentující argumenty, provádí operaci danou konkrétním uzlem a vrací návratovou hodnotu. Implementace interpreteru je relativně snadná a přitom velice efektivní. Jeho celková efektivita je samozřejmě ovlivněna kvalitou nativního překladače jazyka, který je použit pro implementaci interpreteru. Např. efektivita předávání výsledku vyhodnoceného podstromu jako návratové hodnoty funkce závisí na implementaci volacích konvencí a předávání návratových hodnot. Protože se ale tato konstrukce vyskytuje v rozličných algoritmech velice často, je i její překlad zpravidla velice kvalitní (prakticky všechny volací konvence obvyklé v operačních systémech pro procesory x86 vrací návratovou hodnotu v registru EAX, případně RAX, tedy maximální rychlostí bez zpomalení přístupem do paměti).

Pokud uživatel používá OCL (implementuje nějakou proceduru), má k dispozici mohutnější výrazové prostředky, než je tomu v případě pouhé parametrizace komponent s použitím výrazů. Praxe ukazuje, že zejména obdoba podmíněného příkazu bývá při omezení výrazových prostředků pouze na výrazy postrádána. Do jisté míry lze tento nedostatek napravit zabudovanou funkcí iif.

Kupříkladu programová konstrukce řešící omezení definičního oboru funkcí či operátorů:

```
if b = 0 then
  c := 0;
else
  c := a / b;
end;
```

může být s použitím funkce iif nahrazena jediným výrazem:

```
c := iif( b = 0, 0, a / b );
```

Obdoba funkce iif bývá v různých jazycích a nástrojích implementována různě. Např. v Jazyku C i v synchronním jazyku Esterel je její obdobou podmíněný operátor ? :. V jazyce Object Constraint Language (tento jazyk je součástí specifikace UML a je rovněž označován zkratkou OCL) a také v synchronním jazyku Lustre je obdobou funkce iif přímo konstrukce if…then…else užitá přímo v zápisu výrazu.

Jistý problém vyhodnocování výrazů rekurzívním sestupem může představovat omezení velikosti zásobníku. Při vyhodnocení typických výrazů se hloubka zanoření pohybuje v řádu jednotek, v některých případech je ale třeba vyčíslit výraz, jehož stromová reprezentace prakticky degeneruje na seznam (např. dlouhá řada sčítání), a pak spotřeba prostoru na zásobníku může představovat problém.

Zejména v prostředí operačních systémů pro zabudovaná nasazení (např. Windows CE), kdy operační systém nemá k dispozici odkládací soubor, bývá prostor pro zásobník relativně omezený.

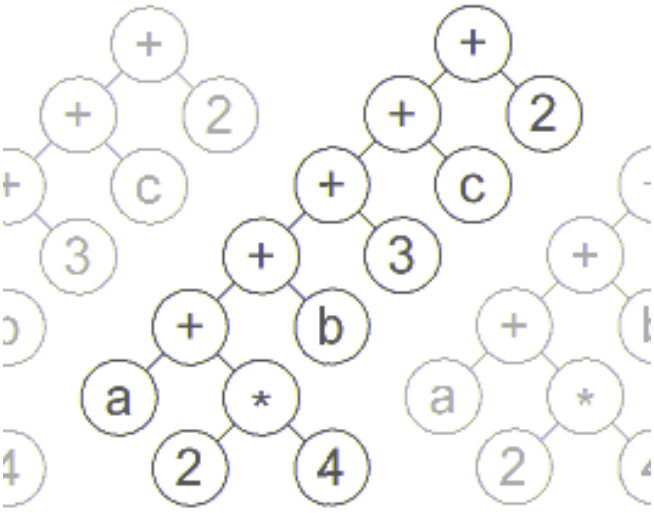
Spotřeba prostoru na zásobníku se ale může velice lišit. Zatímco procedura realizující sčítání dvou čísel na zásobník ukládá pouze návratovou adresu, implementace procedur realizujících vyčíslení složitějších operandů nebo funkcí (např. převody mezi řetězci a čísly apod.) mají spotřebu prostoru na zásobníku podstatně větší. To je důležité mít na paměti při implementaci interpreteru a nespotřebovávat zásobník zbytečně, byť za cenu mírně vyšší režie.

2.2.3. Optimalizace výrazů

Stromová reprezentace je velmi vhodná pro optimalizace. OCL je často využíván pro obsluhu velkého množství dat a tak jeho efektivita v některých případech podstatně ovlivňuje odezvu aplikace. Optimalizace výrazů je tedy důležitou vlastností OCL.

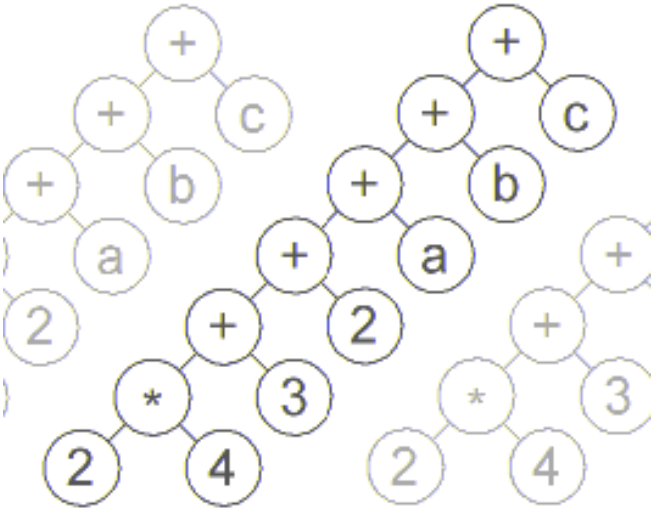
Zcela základní optimalizací je detekce konstantních výrazů – ty jsou vyhodnoceny a nahrazovány konstantou reprezentující výsledek již v době překladu.

Další metodou je detekce konstantních podvýrazů a jejich vyčíslení v době překladu. Stromová struktura se tedy často podstatně liší od zdrojového výrazu. Při hledání konstantních podvýrazů je využíváno komutativnosti operací sčítání a výraz je převeden do podoby nejlépe vyhovující optimalizaci. Např. výraz a + 2 * 4 + b + 3 + c + 2:



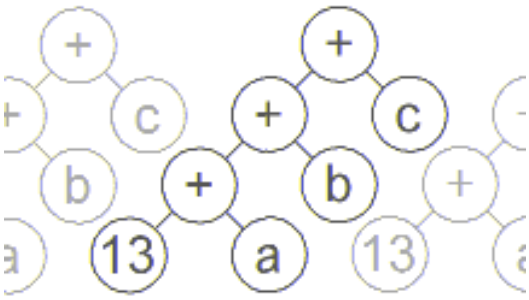
Obr. 2: Stromová reprezentace výrazu

je převeden na výraz 2 * 4 + 3 + 2 + a + b + c:



Obr. 3: Mezistupeň optimalizace výrazu

a optimalizován na 13 + a + b + c:



Obr. 4: Konečná podoba optimalizovaného výrazu

Pokud je to pro optimalizaci výhodné, je odčítání podvýrazu převedeno na sčítání vložením unární negace a dále je podvýraz zpracován jako sčítání.

2.2.4. Detekce vždy pravdivých či nepravdivých výrazů

Překvapivě velké množství uživatelů používá programové konstrukce, které nijak neovlivňující logiku algoritmu, pouze zavádějící nadbytečnou režii. Příkladem může být neopodstatněná podmínka:

```
if a or not a then
…
```

Tato podmínka je samozřejmě vždy vyhodnocena jako pravdivá bez ohledu na hodnotu proměnné a. Překladač tuto situaci detekuje a vypíše varování.

V některých případech je pro uživatele obtížnější trvalou pravdivost podmínky rozpoznat, např. cyklus:

```
index := 10;
while index > 0 do
  …
  index := index – 1;
end;
```

je nekonečný, pokud je „index“ proměnná bezznaménkového typu, jehož hodnota nikdy není menší než 0. Podobně cyklus:

```
index := 0;
repeat
  …
  index := index + 1;
until i > 127;
```

bude nekonečný, pokud je „index“ proměnná typu shortint (znaménkové číslo s rozsahem -128 až 127). I tyto případy překladač detekuje a vypisuje varování.

Specifikou systému Control Web je tzv. „duální programování“. Vývojové prostředí umožňuje nejen editovat zdrojový text, ale také vyvíjet aplikaci v grafickém režimu. V okamžiku překlopení aplikace do grafické podoby zdrojová podoba aplikace zaniká, překladač překládá a kontroluje mimo jiné také zápisy všech výrazů. V grafickém módu vývojového prostředí tedy původní textová podoba výrazů neexistuje a při zpětném překlápění aplikace do textové podoby je vytvářena ze stromové struktury. Z tohoto důvodu jsou veškeré optimalizace podmíněny a provádí se pouze pokud je aplikace překládána za účelem následujícího běhu, nikoliv za účelem grafické editace.

Vedlejším efektem generování textové podoby výrazu ze stromové struktury je tak uspořádání jejich textové podoby – zatímco identifikátory operandů a funkcí jsou samozřejmě rekonstruovány zcela přesně, např. počty mezer není možno rekonstruovat. Dalším důsledkem je vynechání nadbytečných závorek v generovaném textu, pokud je správné pořadí vyhodnocení zajištěno prioritou operátorů.

2.2.5. Saturační a modulo aritmetika

Výsledek vyhodnocení výrazu může být přiřazen do datového elementu, jehož typ není schopen výsledné číslo pojmout a proto musí být nějakým způsobem oříznuto. Zcela intuitivní je pominutí desetinné části čísla v plovoucí řádové čárce při přiřazení do celočíselného typu. Ale i celočíselné typy jsou různě velké a mohou vyžadovat oříznutí. V principu existují dva způsoby, jak číslo přesahující rozsah daného typu pozměnit (oříznout), aby je bylo možno do datového typu zapsat.

- Saturační aritmetika při pokusu zapsat do datového elementu hodnotu mimo rozsah datového typu zapíše největší nebo nejmenší hodnotu v datovém typu reprezentovatelnou (datový element je saturován – nastaven na extrémní hodnotu). Následující příklad ukazuje toto pravidlo:

```
var
  c : shortcard; (* rozsah 0..255 *)
  i : shortint;  (* rozsah -128..127 *)
begin
  c = -1;     (* c = 0 *)
  c = 256;    (* c = 255 *)
  i = -129;   (* i = -128 *)
  i = 128;    (* i = 127 *)
```

- Modulo aritmetika zapíše do datového elementu hodnotu odpovídající zbytku po dělení (zbytek po dělení je získán operací modulo) výsledku maximálním rozsahem typu. V praxi se při výpočtech zbytek po dělení samozřejmě nepočítá, stejného efektu se dosáhne pokud se informace mimo rozsah typu prostě zapomenou. Následující ukázka demonstruje modulo aritmetiku:

```
var
  c : shortcard; (* rozsah 0..255 *)
begin
  c = -2;     (*   11111110, c = 254 *)
  c = 0;      (*   00000000, c = 0 *)
  c = 255;    (*   11111111, c = 255 *)
  c = 256;    (* 1 00000000, c = 0 *)
  c = 400;    (* 1 10010000, c = 144 *)
```

Je obtížné odhadnout, který způsob přiřazení je pro běžného uživatele intuitivnější. Aritmeticko-logické jednotky obecně používaných procesorů tradičně implementují modulo aritmetiku a pro jejich programátory je její použití zcela přirozené. Naproti tomu digitální signální procesory implementují saturační aritmetiku nebo alespoň umožňují programátorovi zvolit způsob ořezání čísel. V oblasti průmyslových automatů je použití saturační aritmetiky rovněž obvyklé.

V systému Control Web byla jako implicitní zvolena saturační aritmetika. Některé algoritmy ale spoléhají na modulo aritmetiku – např. výpočet kontrolního součtu. Aby v takovém případě nebylo nutné ve výpočtu vždy explicitně provádět časově náročnou operaci modulo (operátor %), zabraňující přetečení a tím i saturaci datového elementu, existuje v rámci systému Control Web možnost explicitního použití modulo aritmetiky pro určitý datový element. U datového elementu, u něhož má být použita modulo aritmetika (nebo u celé datové sekce) je nutno definovat atribut „modulo_arithmetics“.

```
var
  c : shortcard { modulo_arithmetics };
begin
  c = 200 + 200; (* c = 144, nikoliv 400 ani 255 *)
```

2.2.6. Reprezentace řetězců a řetězcové výrazy

Datový typ řetězec nebývá obvykle považován za jednoduchý datový typ. Nelze obecně stanovit jeho maximální délku a podpora práce s řetězci nebývá součástí definice jazyka samotného, ale spíše podpůrných knihoven. OCL je ale jazyk vysoké úrovně a podporuje práci s řetězci na úrovni základních datových typů bez omezení délky.

Omezení délky samozřejmě existuje, ale je dáno spíše omezeními správy paměti než principiální implementací. V praxi lze např. s celým souborem délky milionů byte pracovat jako s jediným řetězcem.

Operandy součtu (operátor „+“) a porovnání (operátor „=“) jsou definovány rovněž nad typem řetězec – operátor součtu provádí spojení řetězců. Operace přiřazení pracuje s typem řetězec zcela intuitivně. Další operace (vyhledávání a nahrazování podřetězců apod.) s řetězci jsou implementovány ve formě zabudovaných řetězcových funkcí. Uživatel je tak odstíněn od všech operací spojených a alokováním řetězců, jejich kopírováním apod. Na druhé straně musí implementaci těchto operací a správu řetězců převzít virtuální stroj.

Jak již bylo řečeno, virtuální stroj vyhodnocuje výraz v podobě stromové datové struktury rekurzivně a výsledek každé operace vrací v podobě návratové hodnoty. Pro logické a číselné datové typy to je jeden z nejvýhodnějších způsobů, problém ale nastane s řetězci, zejména pokud jsou implementovány dynamicky (tedy z hlediska uživatele bez omezení délky). Dynamický řetězec není možné předávat jako návratovou hodnotu. Předávání reference na řetězec také není možné, neboť by tím zanikla funkce implicitního zásobníku realizovaná rekurzí.

Dřívější verze jazyka OCL skutečně omezovaly délku řetězce na 256 znaků. Ačkoliv převedení řetězce na statickou strukturu s pevnou délkou usnadní implementaci, důsledkem je plýtvání pamětí na jedné straně (typický řetězec je kratší než 256 znaků) a současně omezení uživatelů na druhé straně (např. při konstrukci SQL dotazu potřebná délka řetězce často přesáhne tento limit). Z těchto důvodů byla v OCL implementována podpora pro dynamické řetězce.

Pro vyhodnocování řetězcových výrazů byla navržena datová struktura vyrovnávací paměti, která plní dvě funkce – jednak implementuje explicitní zásobník, nutný pro

vyhodnocování (návratovou hodnotou procedury realizující řetězcovou operaci je ukazatel na vrchol zásobníku), navíc dokáže recyklovat paměť alokovanou pro řetězce. Při vybírání řetězců ze zásobníku (operace pop) není během vyhodnocování paměť alokovaná pro řetězce vracena systému, ale ponechána pro případnou následující operaci vložení na zásobník (operace push). Daný paměťový blok je realokován pouze pokud se nově vkládaný řetězec do již alokovaného prostoru nevejde. Za běhu aplikace se délka alokovaných řetězců ustálí na skutečné maximální délce používané v aplikaci a poté se již časově náročná operace alokace paměti neprovádí. Tento mechanizmus zajišťuje dokonce lepší výpočetní výkon při interpretaci řetězcových výrazů než je tomu v případě řetězců s pevnou délkou – obsah řetězců není kopírován na zásobník. Také nároky na velikost zásobníku jsou výrazně menší.

2.2.7. Datový typ data

Role OCL jako spojovacího článku mezi komponentami aplikace si vynutila zavedení specifického anonymního datového typu nazvaného „data“. Specifičnost tohoto typu spočívá ve faktu, že jazyk samotný nepodporuje prakticky žádné operace s tímto typem. Je navržen jako datový blok dané délky, který je možno přiřazovat a předávat jako argument do OCL procedur.

Jediné dvě operace nad typem data definované v OCL jsou spojení dvou bloků (operátor +) a porovnání na shodnost případně různost (operátory = a <>). Tyto operace nevyžadují žádné znalosti vnitřní struktury datového bloku. Operace spojení vytvoří nový datový blok o délce rovnající se součtu délek obou operandů a obsahující datový obsah prvního následovaný obsahem druhého operandu. Samozřejmě je operace spojení v řadě případů nesmyslná (např. pokud je obsahem datových bloků JPG obraz, jejich spojením nevznikne nic), pak je na programátorovi aby ji nevyužíval.

Porovnání na rovnost vyžaduje bloky shodné délky (v případě různé délky jsou bloky rozdílné) a porovnává jednotlivé byte datových bloků. Jazyk OCL nedefinuje relaci uspořádání nad tímto typem a proto operace < nebo > nejsou nad tímto typem povoleny.

Význam zavedení typu, jež v samotném jazyce není příliš k užitku, spočívá hlavně v umožnění spolupráce rozdílných komponent. Kupříkladu komponenta HTTP serveru poskytne soubor odeslaný uživatelem z formuláře a předá jej aplikaci jako argument událostní procedury OnPostFile. Tato procedura chce obsah souboru uložit na lokální disk a využije volání procedur Create, Write a Close systémového přístroje „file“. Právě

argument procedury Write je typu data, stejně jako vstupní argument událostní procedury OnPostFile. Ačkoliv je vnitřní struktura souboru algoritmu skryta a jediná informace kterou má kód k dispozici je jeho délka, jednotlivé komponenty již s ní dokáží pracovat.

Nad typem data je definována zabudovaná funkce jazyka „size“, která vrací délku (počet byte) argumentu tohoto typu.

Vnitřní implementací se typ data velmi podobá typu řetězec. Ve starších verzích OCL skutečně dynamický řetězec (typ string) sloužil na místo typu data. Problém vznikne, pokud je OCL použit v UNICODE variantě, kdy každý znak obsahuje 2 byte. Řetězce tedy mají vždy sudý počet byte a neexistuje způsob, jak zachovat délku obecného datového bloku. Pokud je délka datového bloku lichá, je po konverzi do UNICODE řetězce vždy zaokrouhlena na nejbližší vyšší sudé číslo a původní délka bloku je ztracena.

2.3. Procedury – základní bloky OCL kódu

V prostředí Control Web je veškerý kód soustředěn do procedur. Neexistuje žádný „hlavní program“ nebo funkce či statická metoda „main“, jejímž vyvoláním by aplikace startovala. Plyne to ze skutečnosti, že použití OCL je pro tvorbu aplikací v zásadě nepovinné a aplikace je schopna práce i bez jediného řádků OCL kódu.

Procedury nemohou existovat samostatně, nezávisle na nějaké komponentě (virtuálním přístroji či datová sekci). Každá procedura je součástí definice nějakého virtuálního přístroje nebo datové sekce.

V systému Control Web existují tři kategorie OCL procedur:

1. Událostní procedury (event procedures) jsou systémem vyvolávány v okamžiku vzniku dané události. Názvy událostních procedur jsou voleny tak aby napovídaly danou událost (OnActivate, OnOutputChanged, OnMouseMove, apod.).

2. Uživatelské procedury (user procedures) jsou čistě věcí autora aplikace. Systém sám je nevolá a je na kódu událostních procedur nebo jiných uživatelských procedur, aby uživatelské procedury volal.

3. Nativní procedury (native procedures) představují programové rozhraní komponent. Nativní procedury virtuálních přístrojů nejsou v pravém slova smyslu procedurami jazyka OCL. Každý virtuální přístroj může nabídnout mimo

vlastností také řadu procedur s rozhraním odpovídajícím konvencím OCL, avšak implementovaných nativním kódem virtuálních přístrojů. Kód OCL je tedy může vyvolávat podobně jako uživatelské či událostní procedury, jejich implementace ale není součástí kódu aplikace.

Jednou z událostí je start aplikace a s ní spojená událostní procedura OnStartup(), která je během startu spuštěna. Implementace této událostní procedury má tedy zcela shodný význam jako např. implementace funkce „main" v jazyku C – na rozdíl od funkce main() ale může proceduru OnStartup() implementovat v aplikaci systému Control Web každý přístroj. Jsou pak spouštěny sekvenčně podle výskytu patřičných virtuálních přístrojů či datových sekcí v aplikaci. Pro Control Web ale není typické že celý běh aplikace je řízen kódem jediné procedury. I pokud nějaký přístroj (přístroje) implementuje proceduru OnStartup(), kód zpravidla provede inicializaci a proceduru opustí. Celý běh je pak řízen událostmi.

Na různé kategorie procedur v jazyce OCL se lze dívat z několika různých hledisek.

Kategorie procedur	*Je možné ji volat*	*Je implementována ve kódu aplikace*	*Je volána systémem při vzniku události*
Nativní	Ano	Ne	Ne
Událostní	Ano	Ano	Ano
Uživatelské	Ano	Ano	Ne

Tabulka 1: Kategorie procedur v jazyce OCL

2.3.1. Signatura procedury a přetěžování procedur

Signatura (podpis) procedury je používána k jejímu jednoznačnému určení a rozlišení od ostatních procedur. V systému Control Web je signatura procedury tvořena jejím jménem, parametry a návratovou hodnotou. Proto je možné například definovat dvě navzájem rozdílné procedury:

```
procedure SetColor( color : longcard );
procedure SetColor( red, green, blue : shortcard );
```

Ačkoliv se obě procedury stejně jmenují, každá má jiné parametry a tak není problém je při volání rozlišit. Pokus o zavolání „SetColor" s jiným počtem argumentů povede k chybě při překladu.

Možnost definovat dvě procedury stejného jména, lišící se jen typy a/nebo počty argumentů bývá označována jako přetěžování procedur. Předchozí příklad demonstruje

výhodnost tohoto konceptu např. pro nastavení barvy – barva může být zadána jako jediné číslo či jako tři separátní barevné složky.

Rozlišení typů argumentů je ale ovlivněno automatickou konverzí číselných typů. Jejím důsledkem je, že signatury procedur se stejným počtem byť rozličných číselných parametrů jsou zaměnitelné a nelze je tedy současně definovat u jednoho přístroje. Pak definice:

```
procedure SetColor( color : integer );
procedure SetColor( color : real );
```

způsobí chybu „procedura redefinována“.

2.3.2. Kódování signatur a implementace pozdní vazby

Aby mohly být jednotlivé komponenty řízeny kódem v jazyce OCL, musí nabízet své nativní procedury pro volání z jiných částí aplikace. V jazyce OCL je vazba ustavena virtuálním strojem v okamžiku startu aplikace. K ustavení vazby tedy nedochází až v okamžiku volání, jako je tomu u dynamických jazyků. Na druhé straně není vazba ustavována při překladu a není tedy nutné, aby v době překladu byla k dispozici typová informace všech komponent. Vyřešení vazby v okamžiku startu aplikace výrazně zmenšuje režii volání procedury a zajišťuje deterministickou dobu volání za běhu, což je pro aplikace reálného času podstatné. Především ale zajišťuje kontrolu správnosti spouštěné aplikace při startu a vylučuje tak možnost vzniku neošetřených výjimek za běhu aplikace. Protože aplikace řídící průmyslové procesy pracují typicky bez dohledu, je ohlášení chyby „volání neexistující procedury“ za běhu, kdy operátor není přítomen nebo nemá možnost jednoduché nápravy, nepřípustné.

Ustavení vazby začíná identifikací objektu daného jména. Mechanismy správy objektů jsou součástí jádra systému Control Web a virtuálnímu stroji OCL jsou k dispozici prostřednictvím API funkcí.

Překladač z přečteného zápisu volání vytvoří signaturu volané metody v podobě textového řetězce. Řetězec je tvořen podle následujícího pravidla:

```
Signature → Procedure_name # { type_letter } [ # type_letter ]
```

Řetězec signatury je tvořen jménem procedury, následovaný kódovanou typovou informací popisující parametry procedury, oddělenou speciálním znakem ‚#‘. Pokud procedura nemá parametry ani návratovou hodnotu, je znak ‚#‘ poslední znak signatury. Typ návratové hodnoty je kódován podobně jako typy parametrů, je ale uvozen dalším znakem #, a to i

v případě, že procedura nemá parametry. Pak je typ návratové hodnoty umístěn za dvěma znaky ##. Například signatura procedury:

```
procedure CheckName()
```

je textový řetězec:

```
CheckName#
```

Stejná procedura s návratovou hodnotou:

```
procedure CheckName(): boolean
```

má signaturu:

```
CheckName##l
```

Typy parametrů a návratové hodnoty jsou kódovány podle tabulky:

Datový typ	*Kód typu*
undefined	?
boolean	l
shortcard	b
cardinal	c
longcard	C
shortint	j
integer	i
longint	I
shortreal	r
real	R
strng	s
data	d
any	*

Tabulka 2: kódování typů v signatuře OCL procedur

Pak například signatura procedury:

```
procedure Calculate( x, y : real; flag : boolean ): string;
```

bude zakódována do řetězce:

```
Calculate#RRl#s
```

Jazyk OCL dovoluje předávat proměnné odkazem. Protože volací konvence je klíčová pro implementaci procedury, je rovněž kódována v sigantuře pomocí znaku ‚&' uvedeného před znakem kódujícím typ parametru předávaného odkazem:

```
procedure Calculate( var x, y : real; flag : boolean ): string;
```

bude zakódována do řetězce:

```
Calculate#&R&Rl#s
```

Další specifickou vlastností jazyka OCL je schopnost předávat odkaz na celá pole. Uvnitř kódu procedur je možné přistupovat k poli předanému parametrem stejně jako k jiným globálním polím.

Určitým problémem je získávání mezních hodnot indexů polí předaných do procedur. Přístup k prvku pole mimo deklarovaný rozsah vede k chybě za běhu aplikace. Pokud je chyba potlačena, vede alespoň k selhání přístupu k prvku pole. Když algoritmus pracuje s globálním polem, je možné např. deklarovat rozsahy pole s použitím konstant a tyto konstanty rovněž použít v algoritmu s polem pracujícím. Má-li ale algoritmus v proceduře pracovat s obecným polem, je velmi užitečné zabudovat do jazyka mechanismy získávání mezních indexů polí. V OCL jsou k dispozici dvě zabudované funkce „loindex“ respektive „hiindex“, jejichž argumentem je pole (tedy nikoliv prvek pole) a návratovou hodnotou je nejnižší respektive nejvyšší index daného pole. Procedura v jazyce OCL hledající maximální hodnotu v poli tak může vypadat např. následovně:

```
procedure Extremes( a : array of real; var min, max : real );
var
  l : integer;
  i : integer;
begin
  l := loindex( a );
  min := a[ l ];
  max := min;
  for i := l + 1 to hiindex( a ) do
    if a[ i ] > max then
      max := a[ i ];
    elsif a[ i ] < min then
      min := a[ i ];
    end;
  end;
end_procedure;
```

Pole jsou předávána vždy odkazem, nikdy nejsou kopírována na zásobník. Protože neexistuje jiná možnost, klíčové slovo „var“ před parametrem typu pole je nepovinné. Předávání polí tak představuje specifickou volací konvenci a rovněž je kódováno v řetězci popisujícím signaturu procedury – parametry typu pole jsou uvozeny znakem ‚[‘. Signatura výše zmíněné procedury tedy vypadá následovně:

```
Extremes#[R&R&R
```

Virtuální stroj zkonstruuje textové reprezentace signatur volaných metod během překladu na základě znalosti typů argumentů předávaných do metody. Při spouštění aplikace virtuální stroj získá odkaz na instanci volaného objektu uchová ji v mezikódu pro použití za běhu. Poté instanci volá metodu „SignatureToHandle“. Pokud daná instance objektu

signatuře metody porozumí, vrátí handle, jimž si přeje při volání metodu identifikovat. Pokud ne, virtuální stroj oznámí chybu při startu aplikace „volání neznámé procedury“.

Za běhu je volání metody realizováno voláním metody „Invoke“ s handle procedury vráceným z volání „SignatureToHandle“.

> *Součástí rozhraní komponent v systému Control Web jsou mechanismy dovolující vyčíslit všechny procedury včetně názvů, popisů a typů parametrů. Tato rozhraní jsou používána v integrovaném uživatelském prostředí k tvorbě nápovědy pro uživatele, v inspektoru přístroje k tvorbě nabídek procedur, jež má komponenta k dispozici apod. Při překladu a běhu aplikace ale nejsou tyto mechanismy využívány a spoléhá se pouze na rozhraní SignatureToHandle a Invoke. Verze systému určená pro spouštění hotových aplikací (runtime verze) proto tato rozšířená API ani neobsahuje.*

Zcela shodných mechanismů využívá virtuální stroj i při ustanovení vazeb mezi uživatelskými procedurami. Jediný rozdíl spočívá ve skutečnosti, že uživatelské procedury nevyžadují ze strany objektu, jemuž jsou volány, žádné ošetření – vše řeší implementace metod SigantureToHandle a Invoke na úrovni abstraktního předka komponenty.

2.3.3. Signatury nativních procedur

Signatury procedur psaných v OCL jsou pevně dané, nicméně signatury nativních procedur jednotlivých komponent mohou být bohatší. Jak již bylo řečeno, nativní procedury nejsou implementovány v OCL, ale v programovacím jazyce užitém k implementaci dané komponenty. Mají tedy přístup k datovým strukturám argumentů na nižší úrovni než kód OCL. Kód implementující nativní procedury vidí argumenty jako pole variantních záznamů včetně datových položek rozlišujících typ varianty.

V nativní proceduře můžeme definovat argument písmenem ‚n‘, které je interpretováno jako libovolný číselný typ. Při konstrukci datové struktury předávající argumenty tedy není nutné konvertovat numerické typy aby odpovídaly typům v signatuře. Pokud nativní procedura argument označí jako ‚n‘, přebírá sama zodpovědnost za konverze uvnitř svého kódu. Uživatel pak v popisu nativní procedury vidí typ popsaný jako „number“.

V určitých případech je žádoucí, aby nativní procedura akceptovala parametr jakéhokoliv typu. Znak ‚*‘ je zástupný pro jakýkoliv typ a v popisu nativní procedury je uveden typ „any“. Je na kódu nativní procedury, aby zjistila typ předaného argumentu a pracovala

s ním patřičným způsobem. Pokud nativní procedura deklaruje schopnost práce s jakýmkoliv typem, pak předání jakéhokoliv typu nesmí způsobit chybu za běhu aplikace.

Příkladem použití zástupného datového typu je nativní procedura DebugOutput systémového přístroje „core“, sloužící k vypsání hodnoty datového elementu. Signatura této procedury je:

```
DebugOutput#s*
```

Po zavolání této procedury se v okně s ladicími výpisy se objeví řetězec předaný jako první argument následovaný textovou reprezentací libovolného datového elementu předaného jako druhý argument.

2.3.4. Vazby událostí a procedur

Mechanismus událostních procedur je poměrně obvyklý u komponentových systémů a skriptovacích jazyků. Jednotlivé systémy se ale podstatně liší způsobem ustavení vazby mezi událostí a procedurou (funkcí), která je při této události vyvolávána – jednou z možností je např. direktiva programovacího jazyka „event“ apod.

Množina událostí je definována komponentou samotnou. I v případě systému Control Web jednotlivé virtuální přístroje a datové sekce definují množinu událostí při jejichž výskytu je vyvolán kód svázané procedury. Vazba je provedena pouze na základě signatury (podpisu) procedury. Komponenta tedy „předepíše“ signaturu a pokud nějaká procedura této signatuře odpovídá, je považována za událostní.

Mimo několika systémových událostních procedur (např. procedura OnActivate() je vyvolávána vždy při aktivaci přístroje) definuje každý přístroj množinu vlastních událostních procedur, které bude přístroj vyvolávat, pokud je autor aplikace implementuje (např. přístroj hlídající mezní stavy hodnot vyvolává proceduru OnAlarm() apod.).

Ustavení této vazby probíhá rovněž v okamžiku startu aplikace a používá identických mechanismů, jako je tomu u ustanovení vazeb při volání nativních procedur objektů, pouze původci volání jednotlivých metod se liší. Jak již bylo řečeno, událostní procedura se v jazyce OCL syntakticky neliší od běžné uživatelské procedury, pouze její signatura je „známa“ dané komponentě. Při startu aplikace tedy komponenta volá metodu SignatureToHandle svému abstraktnímu předkovi, který spravuje všechny ne-nativní procedury (tedy procedury, jejichž zápis je uveden ve zdrojovém kódu aplikace), a předává mu signatury svých událostních procedury. Pokud je v aplikaci u dané komponenty

procedura s uvedenou signaturou implementována, metoda SignatureToHandle vrátí platný handle. Je na implementaci konkrétní komponenty, aby handle uchoval a při výskytu dané události zavolal metodu Invoke s patřičnými parametry, odpovídajícími dané signatuře.

Použití mechanizmu signatur je na jedné straně velice intuitivní a snadno použitelné (není zaváděno žádné nové klíčové slovo, syntaxe jazyka není rozšiřována), na druhé straně ale ze zdrojového textu neplyne, zda se jedná u proceduru událostní nebo uživatelskou. Pokud není signatura dodržena (např. není dodržena velikost písmen v názvu procedury), procedura bude považována za uživatelskou a nebude při výskytu události volána přestože je uživatel přesvědčen o opaku. Z těchto důvodů vývojové prostředí indikuje, zda-li byla signatura dodržena a procedura je svázána s danou událostí (a bude tedy při výskytu události volána), či nikoliv (její signatura neodpovídá žádné události). Při implementaci navíc vývojové prostředí nabízí přímo připravené hlavičky událostních procedur u každé komponenty a tím se problémy s vytvořením správné signatury prakticky eliminují.

2.4. Abstrakce přístupu k datům – asynchronní rozhraní ovladačů periferních zařízení

Hardware používaný v průmyslové automatizaci je velice různorodý – podle konkrétní aplikace se používají jednotky distribuovaných vstupů a výstupů, průmyslové automaty, měřicí karty apod. Vyrábí jej řada firem, počínaje lokálními firmami s několika zaměstnanci a konče nadnárodními společnostmi. Stejně jako průmyslový hardware jsou různorodé i způsoby komunikace mezi průmyslovými jednotkami a řídicími počítači.

Poměrně značná setrvačnost celého oboru je příčinou stálého rozšíření komunikačních standardů, které byly ve světě kancelářských a domácích aplikací již nahrazeny modernějšími standardy – např. sériová linka RS-232C byla nahrazena sběrnicí USB, řada průmyslových zařízení ji však stále používá.

V technologiích je velice často nutné rozmístit vstupně/výstupní moduly v relativně velkých vzdálenostech (až stovky metrů) od řídicího počítače (rozvaděče). Ačkoliv sériová linka má dosah jen desítky metrů, často je používán převodník na rozhraní RS-485, využívající diferenční proudovou smyčku s dosahem stovek metrů. Takové vzdálenosti periferií nejsou v kancelářském nasazení obvyklé a proto se i používaná rozhraní liší. S ústupem sériového rozhraní ale i v průmyslu roste využití moderních rozhraní, např. USB s prodlužovacími členy či lokální síť Ethernet.

Každý programový systém, který si klade za cíl být obecným vývojovým nástrojem pro průmyslové aplikace a nikoliv jen programovou podporou produktů konkrétního výrobce, musí být schopen se všemi velice různorodými zařízeními komunikovat. Ačkoliv se objevují snahy standardizovat programové rozhraní průmyslových jednotek (např. OPC – OLE for Process Control – standard přístupu k datům založený na technologii Microsoft COM), nelze na ně zcela spoléhat, protože jejich rozšíření není dostatečně univerzální. Různorodost způsobů komunikace je nutno postihnout vlastní programovou vrstvou.

Synchronní (blokující) přístup k datům v technologii představuje „naivní“ přístup, který v praktických nasazeních není k užitku.

> *Pojem synchronnosti nyní vztahujeme k aplikaci samotné. I pokud aplikace přistupuje např. k sériovému kanálu synchronně (tedy prostřednictvím blokujícího volání), operační systém sám zajistí pozdržení volajícího prováděcího toku a případně spustí jiný připravený prováděcí tok. Celková průchodnost operačního systému tak není blokujícím voláním omezena, nám se ale jedná o průchodnost konkrétní aplikace.*

V principu je činnost řídicí aplikace smyčka, která čte vstupní data, zpracovává je a poté zapisuje výstupní data. Toto schéma je ale zjednodušením na hranici únosnosti, neboť množina načítaných dat je obecně v čase proměnná, stejně tak se mění jejich zpracování v závislosti na akcích operátora, stavu technologie apod. Uvnitř smyčky se algoritmy větví a případně zanořují do smyček nižší úrovně atd. Zjednodušení ale dobře ilustruje skutečnost, že řídicí aplikace je často závislá na datech přečtených z technologie a algoritmus zpravidla nemůže pokračovat zápisem výstupů, pokud nemá k dispozici spolehlivé vstupní hodnoty.

Každý komunikační cyklus je v systému Control Web rozložen do několika fází. Jediný cyklus probíhá v každém systémovém časovém kroku (systémový časový krok nastane v okamžiku, do něhož padne aktivace alespoň jedné datové sekce, či jednoho virtuálního přístroje v aplikaci).

1. Nejprve všechny datové sekce a virtuální přístroje, aktivované v daném systémovém časovém kroku, označí komunikované datové elementy, jejichž hodnoty budou v rámci své aktivace vyžadovat.
2. Komunikační vrstva systému Control Web označí tyto elementy ovladačům periferních zařízení, které zprostředkovávají jejich komunikaci. Samotné

označení ale ještě nezpůsobuje jejich komunikaci, každý ovladač si pouze poznamená, o které datové elementy je v daném časovém kroku zájem.

3. Jakmile jsou všechny datové elementy označeny, komunikační vrstva informuje ovladače, že v daném časovém kroku nebudou již další datové elementy komunikovány. Ovladače nyní mají kompletní informaci o potřebných kanálech a mohou zahájit komunikaci. V závislosti na přenosovém médiu a komunikačním protokolu každý ovladač sestaví požadavky a odešle je do periferií. Kumulace požadavků má zřejmou výhodu – pokud např. čteme osm logických hodnot reprezentovaných v periferii jednotlivými bity v jednom byte, přenos jediného byte je jistě efektivnější, než osm samostatných dotazů.
4. Bezprostředně poté se komunikační vrstva systému Control Web dotáže ovladačů, zda-li jsou požadované kanály k dispozici.
    a. Existuje třída periferií, u nichž je doba získání hodnot prakticky totožná s veškerou režií spojenou s voláním služby jádra operačního systému a přístupem na periferii – např. ovladače měřicích karet na PCI sběrnici při požadavku na získání hodnoty zavolají službu ovladače v režimu jádra operačního systému, která přečte registr (registry) vstupně/výstupní karty. Tato operace je prováděna synchronně a ovladač má hodnoty k dispozici „bezprostředně“ po zahájení čtení.
    b. Typicky ale nemá ovladač hodnoty požadovaných kanálů k dispozici. I pokud komunikuje po rychlé USB sběrnici, doba požadavek/odpověď (označovaná Round Trip Time – RTT) činí několik milisekund. Proto ovladač odpoví komunikační vrstvě, že požadovaná data nejsou k dispozici. Od tohoto okamžiku zůstává komunikační vrstva pasivní a je na ovladači, aby ji zpětným voláním upozornil, že buď dokončil komunikaci daného kanálu nebo došlo k chybě a hodnotu nelze přečíst. Pokud ovladač není schopen rozpoznat např. odpojení periferie, musí definovat maximální prodlevu a po ní informovat komunikační vrstvu o chybě.
5. Pokud ovladač nemá data okamžitě k dispozici, další chování záleží na nastavené maximální prodlevě komunikace (timeout komunikačního kanálu).

a. Pokud je tato prodleva nastavena na 0 sekund, komunikační vrstva předá řízení plánovači časových kroků a ten pokračuje v aktivaci sekcí a virtuálních přístrojů. Při čtení komunikovaných datových elementů je vrácena hodnota obsažená při naposledy realizovaném čtení. Pokud je např. virtuální přístroj časován každou sekundu a komunikace používaného datového elementu zabere 0,5 s, pak je použita vždy hodnota z předešlé sekundy. Komunikace vyvolaná v aktuálním časovém kroku bude během 0,5 s dokončena a přečtená hodnota se použije v následujícím časovém kroku.

b. Pokud je prodleva nenulová, představuje dobu, po kterou komunikační vrstva čeká na dokončení komunikací. Během této doby mají ovladače možnost oznámit dokončení komunikace. V takovém případě pracují virtuální přístroje a datové sekce s aktuálními daty. Pokud se to nepodaří (např. došlo k chybě na komunikační lince a ovladač je nucen opakovat dotaz), aplikace pokračuje v běhu s daty z předešlé aktivace podobně jako v předešlém případě.

Volba komunikačních prodlev v praktickém nasazení velice záleží na aplikaci a řízené technologii. Pokud je např. frekvence čtení a zpracování dat alespoň dvojnásobná než vyžaduje řízený proces, je možné v aplikaci pracovat s hodnotou z předešlého časového kroku bez újmy na kvalitě řízení a regulace. Pokud tomu tak ale není, je nastavení komunikačních prodlev velice důležité. Control Web umožňuje mimo statického také dynamické nastavování těchto prodlev a rovněž poskytuje pomocí atributů komunikovaných datových elementů aplikaci informace o aktuální prodlevě, o stavu komunikace (byla-li dokončena do vypršení prodlevy apod.), o chybových stavech apod. Tato problematika ale souvisí spíše s praktickými aplikacemi a nekryje se se zaměřením této práce.

2.5. Dopředné vyhodnocování a optimalizace komunikací

Komunikované datové elementy (kanály, elementy vzdálených modulů apod.) se ve výrazech v OCL vyskytují zcela volně spolu s lokálními datovými elementy. Jak již bylo řečeno, hodnoty komunikovaných datových elementů ale nejsou okamžitě k dispozici a je nutno je získat z ovladačů průmyslových zařízení nebo ze vzdálených spolupracujících počítačů.

Dopravní zpoždění je tedy základním limitujícím faktorem. Naivní přístup spočívající v synchronní komunikaci každého vzdáleného datového elementu by vedl k neakceptovatelným prodlevám. Virtuální stroj tedy musí být navržen tak, aby:

1. Přenášel co největší množství datových elementů v jediném časovém kroku.
2. Nepřenášel žádná data, pokud nejsou v daném časovém kroku zapotřebí.

2.5.1. Optimalizace sdružováním požadavků

Přenos maximálního možného množství dat je důležitý zejména kvůli eliminaci režie přenosového protokolu. Typicky probíhá komunikace se vstupně/výstupní jednotkou podle schématu: odeslání požadavku na měření dat – příjem odpovědi spolu s daty – potvrzení přijetí. Skutečně přenášené množství dat je vždy větší o režii protokolu (pakety inicializace transakce, identifikační data, potvrzení, kontrolní součty apod.).

Efektivita protokolu je přitom závislá na množství přenášených dat. K přenosu osmi logických hodnot stačí komunikovat jediný byte, přitom jediná transakce zahrnuje desítky byte režijních přenosů. Efektivita se v takovém případě pohybuje v řádu pouhé jedné desetiny. Pokud ale na místo přenosu všech osmi logických hodnot v jediné transakci vyvoláme osm transakcí k přenosu jednotlivých bitů, naroste objem přenášených dat i doba komunikace osmkrát.

> *Efektivita je často ovlivňována i jinými mechanismy, které nelze vyvodit z prostého součtu objemu přenášených dat. Jako příklad můžeme uvést přenos dat po sběrnici USB. Sběrnice USB je velice rychlá a teoreticky dokáže přenést až 1MB čistých dat za sekundu (zde máme na mysli USB verzi 1.1, verze 2.0 rychlost přenosu ještě dramaticky navyšuje). Ovšem přenos po sběrnici je řízen v rytmu časových rámců dlouhých 1 milisekundu. Pokud USB zařízení přenáší data do počítače a na požadavek bulk přenosu na vstupním bodu odpoví NAK (Not Acknowledge), počítač bude zařízení opět dotazovat až v následujícím rámci. Tedy i pokud přenos několika desítek byte, nutný ke komunikaci osmi logických hodnot, je teoreticky přenositelný ve zlomku milisekundy, ve skutečnosti může přenos trvat až 8 milisekund.*

Z uvedeného je zřejmá důležitost kumulace všech požadavků na komunikaci v rámci každého časového kroku.

2.5.2. Optimalizace eliminací nepotřebných přenosů

Řada aplikací průmyslové automatizace pracuje s datovými objemy, jejichž kompletní přenos v požadovaných časových intervalech převyšuje možnosti komunikačních linek. Pro správnou funkci aplikace (dodržení požadovaných časových intervalů) je nutné komunikaci přizpůsobit podle potřeb řídicího algoritmu.

Uveďme příklad jednoduché podmínky:

```
if value > 0 then
  a1 := c1a;
  a2 := c2a;
  …
else
  a1 := c1b;
  a2 := c2b;
  …
end;
```

Je zřejmé, že v jediném průchodu algoritmu nebudou zapotřebí kanály c1a, c2a, … a c1b, c2b, … současně. Úkolem virtuálního stroje je komunikovat kanály c1a, c2a, … nebo c1b, c2b, … v závislosti na hodnotě proměnné „value“, nikoliv však oba bloky současně.

2.5.3. Rozhodovací body virtuálního stroje

Jazyk OCL je překládán při startu aplikace do mezikódu, který je poté interpretován virtuálním strojem. Na rozdíl od obecně rozšířených virtuálních strojů neprochází virtuální stroj OCL kód při interpretaci v jediném, ale ve dvou průchodech. Fáze průchodu virtuálním kódem jsou označovány „příprava“ a „běh“.

- Ve fázi „příprava“ je nutné komunikačnímu jádru sdělit, hodnoty kterých datových elementů budu k vykonání kódu zapotřebí. Jádro systému pak může iniciovat komunikaci s ovladači zařízení.
- Ve fázi „běh“ jsou pak potřebné hodnoty již přeneseny (pomineme-li situaci, kdy se nepodařilo hodnoty přenést do zvolené prodlevy) a výrazy mohou být vyhodnoceny.

Těmito fázemi prochází v rámci každého časového kroku všechny virtuální přístroje a také všechny událostní procedury „OnActivate“ v aplikaci. V případě virtuálních přístrojů je označení potřebných datových elementů poměrně elementární. Každý virtuální přístroj má uloženy odkazy na všechny datové elementy, použité ve výrazech zadaných jako parametry přístroje. V rámci fáze „příprava“ každý virtuální přístroj oznámí všechny komunikované datové elementy jádru, aby mohlo zahájit jejich komunikaci. Protože tak učiní všechny

virtuální přístroje aktivované v patřičném časovém kroku, může komunikační jádro spolu s ovladači kumulovat požadavky a optimalizovat přenos.

Virtuální stroj nemůže zahájit přenos všech komunikovaných datových elementů použitých ve spouštěné proceduře bez ohledu na kontext jejich výskytu. Proto ve fázi „příprava“ postupuje kódem a oznamuje komunikačnímu jádru, které datové elementy jsou použity ve výrazech spjatých v jednotlivých instrukcích. Průchod kódem je zastaven pokud:

1. Je nalezena instrukce podmíněného skoku. Instrukce podmíněného skoku jsou v OCL tři:

   - skok pokud je výraz pravdivý
   - skok pokud je výraz nepravdivý
   - skok podle tabulky

   Všechny tyto instrukce obsahují výraz, který určuje, kterou větví se bude kód nadále ubírat (viz. příloha B). Dokud není výraz vyhodnocen, není možné o dalším postupu rozhodnout. Aby bylo možno výraz vyhodnotit, je nutné znát hodnoty dosud komunikovaných datových elementů. Virtuální stroj proto průchod ukončí (naposledy označené datové elementy jsou elementy výrazu podmínky) a nechá jádro zahájit komunikaci. Po dokončení komunikace nastane druhá fáze interpretace OCL kódu „běh“. V této fázi se výrazy skutečně vyhodnocují a výsledky se přiřazují datovým elementům. Poslední vyhodnocený výraz je výraz podmíněné instrukce. Po jeho vyhodnocení nastává opět fáze „příprava“, tentokrát již ve známé větvi kódu.

2. Je nalezena instrukce označená překladačem (označení je realizováno vyhrazeným bitem v instrukčním kódu). Překladač označí instrukce, s nimiž je spojený výraz obsahující komunikovaný datový element (kanál, vzdálená proměnná, …) typu pole s proměnným (nekonstantním) indexovým výrazem. V takovém případě je nutné indexový výraz vyhodnotit, aby bylo možno označit pro komunikaci pouze potřebný prvek komunikovaného pole. Indexový výraz může záviset na libovolném dříve označeném a dosud nekomunikovaném datovém elementu. Je tedy opět nutné fázi „příprava“ ukončit, nechat jádro přenést všechny dosud označené datové elementy, vyhodnotit indexový výraz a označit pro komunikaci pouze potřebný prvek.

Pokud během fáze „příprava“ nedojde k označení žádného komunikovaného datového elementu (žádný se v algoritmu nevyskytuje), virtuální stroj pak ihned provede fázi „běh“, aniž by se zdržoval komunikace s ovladači.

Výše uvedená implementace dovoluje uživatelům volně v kódu používat lokální i vzdálené (komunikované) datové elementy, aniž by bylo nutno zabývat se jejich čtením ze vstupně/výstupních zařízení.

Tento mechanismus přináší i některé nepříjemné důsledky. Uvažme cyklus kopírující data z pole kanálů do pole proměnných:

```
for i := 0 to 15 do
  a[i] := c[i];
end;
```

Cyklus je přeložen do instrukcí virtuálního stroje následovně:

```
          SetVar          a, 0
          JumpTrue        a > 15, exit
label:    SetArray        a[i], c[i]
          JumpTrue        a >= 15, exit
          SetVar          a, a + 1
          Jump            label
exit:
```

Překlad nutně musí obsahovat podmíněnou instrukci vynucující ukončení fáze „příprava“ a vyhodnocení ve fázi „běh“ a to v každém průchodu cyklem. Překladač nedokáže algoritmus analyzovat takovým způsobem, aby v jediném kroku přípravy označil ke komunikaci kanály c[0] až c[15] a poté s nimi pracoval v jediném kroku „běh“. Problém je, že v rámci fáze „příprava“ byl ke komunikaci označen kanál c[i] a jádro musí požádat ovladač o jeho změření. Celá výhoda slučování komunikačních požadavků do bloků je touto konstrukcí eliminována.

Protože přenos bloku dat z periferií do lokálního pole proměnných je velice častá operace (rozhraní průmyslových automatů často zahrnuje pouze definici bloků paměti a přenosy těchto bloků nebo jejich částí), byl do OCL zaveden specializovaný příkaz „move“ a do virtuálního stroje instrukce pro přenos celých polí případně jejích oblastí. Výše zmíněný cyklus je tedy možné zapsat na jediném řádku:

```
move c[0], a[0], 15;
```

Tento příkaz je přeložen do jediné instrukce virtuálního stroje iMove. Při interpretaci instrukce má virtuální stroj má k dispozici všechny potřebné informace (počáteční indexy a počty prvků), aby všech 16 kanálů označil ke komunikaci v jediném průchodu fází

„příprava“ a po dokončení komunikace všechny hodnoty přenesl v opět jediném kroku „běh“.

2.6. Synchronizace s reálným časem

Procedury jsou svázány s konkrétní komponentou aplikace a jsou zpravidla spouštěny jako reakce na výskyt patřičné události. Např. událostní procedura OnActivate periodicky aktivovaného přístroje je spouštěna rovněž periodicky, vždy když je daný přístroj aktivován. Spouštění procedury tedy probíhá v návaznosti na reálný čas.

OCL ale zahrnuje sadu příkazů k synchronizaci algoritmu s reálným časem i v rámci těla procedury. Tyto příkazy bývají označovány jako „zpožďující příkazy“ (ačkoliv jejich použití nemusí nutně znamenat zpoždění vykonávání programu).

2.6.1. Zpožďující příkazy

Zpožďující příkazy jsou čtyři: „wait“, „pause“, „yield“ a „commit“. Poslední příkaz commit není v pravém smyslu slova zpožďující, pouze vynutí komunikaci s periferiemi v rámci právě probíhajícího časového kroku. Jak již bylo uvedeno v předchozí kapitole, elementární jednotkou komunikace je systémový časový krok a žádný kanál není komunikován vícekrát v jediném časovém kroku. Pokud to je zapotřebí, příkaz commit komunikaci vyvolá:

```
output = true;
commit;
output = false;
```

vyšle na výstupní logický kanál output puls. Pokud by příkazy přiřazení nebyly proloženy příkazem commit, na výstup by byla zapsána pouze poslední zapsaná hodnota (false) a to až po ukončení právě probíhajícího systémového časového kroku.

Příkaz „wait“ způsobí aktivní čekání na splnění nějaké podmínky. Jeho syntaxe je jednoduchá – za klíčovým slovem wait následuje podmínka, na jejíž splnění příkaz čeká. Např.:

```
wait level > 10;
```

Pokud je level komunikovaný datový element, bude jeho hodnota přečtena z patřičného ovladače před každým vyhodnocením podmínky.

Pokud je „nekonečné“ čekání nebezpečné, lze příkaz wait zkombinovat s maximální dobou čekání (timeout):

```
wait level > 10, 0.5;
```

Příkaz „pause“ způsobí vložení prodlevy definované délky:

```
pause 0.5;
```

Příkaz „yield“ je jen alternativou příkazu pause s nulovou prodlevou. Tedy zápisy „yield“ a „pause 0“ jsou ekvivalentní.

Jaký význam má uvedení příkazu „pause 0“, že je pro něj dokonce vyhrazeno speciální klíčové slovo jazyka? Příkazy wait, pause a yield velmi úzce souvisí s mechanismy aktivace komponent a také s mechanismy komunikace datových elementů. Za normálních okolností je kód v proceduře vždy vykonán v rámci jediného časového kroku. Během tohoto kroku jsou dříve popsaným způsobem načítány vzdálené datové elementy a rovněž jsou zapisovány výstupní kanály. Datový element, který byl v daném časovém kroku už přečten, není podruhé komunikován. Pak kupříkladu programová smyčka testující kanál level:

```
while level <= 10 do
end;
```

neproběhne buď vůbec nebo bude nekonečná. Kanál level bude přečten v daném časovém kroku jednou (buď při vstupu do smyčky nebo již byl v daném časovém kroku přečten jiným přístrojem) a dále není komunikován.

Má-li se hodnota kanálu level vždy před každým průchodem smyčkou opět komunikovat, je nutno ukončit vykonávání kódu procedury v daném časovém kroku. Příkazy wait, pause a yield přerušení v daném časovém kroku provedou a zajistí návrat k rozpracované proceduře v nejbližším následujícím systémovém časovém kroku (v případě příkazů wait a pause 0 neboli yield), případně za definovanou dobu (po příkazu pause s nenulovou prodlevou). Má-li tedy předchozí příklad pracovat správně, musí její zápis vypadat následovně:

```
while level <= 10 do
  yield;
end;
```

Výše popsané mechanismy mohou způsobovat určité problémy, pokud jsou použity neuváženě. Zejména pokud je procedura volána z jiné procedury, první výskyt zpožďujícího příkazu se z hlediska volajícího kódu jeví jako ukončení procedury. Volání se vrátí do volajícího kódu, aniž by algoritmus mohl doběhnout, se všemi důsledky – hodnoty návratových proměnných budou mít hodnoty z okamžiku přerušení, pokud procedura

například generuje dynamickou HTML stránku, stránka bude odeslána ve stavu, v jakém je v okamžiku přerušení apod.

Procedury přerušené zpožďujícími příkazy jsou v následujících časových krocích vykonávány podobnými mechanismy, jakými jsou aktivovány přístroje (tedy nezávisle na volajícím kódu) a dobíhají již samostatně.

Aby se zamezilo problémům v aplikacích způsobených neuchopením těchto mechanismů uživateli, je výskyt zpožďujících příkazů standardně povolen pouze v událostní proceduře OnActivate. Zpožďování této procedury je intuitivní a rovněž tato procedura není zpravidla volána jiným kódem, ale jen jako reakce na událost aktivace přístroje. Pokud k aktivaci přístroje dochází častěji než je např. délka prodlevy uvedené v příkazu pause, jsou tyto aktivace (volání OnActivate) spotřebovány v rámci čekání na uplynutí intervalu a procedura není startována vždy od počátku. Nový start procedury OnActivate od počátku nastane až v rámci první aktivace následující po úspěšném dokončení předchozího běhu této procedury.

Control Web dovoluje použití zpožďujících příkazů volně ve všech procedurách, je ale nutnou tuto možnost explicitně povolit a autor aplikace musí vít důsledky těchto mechanismů na vědomí. Pokud je například příkaz „pause 1“ uveden v událostní proceduře OnClick přístroje „switch“ a uživatel klikne na přístroj 2× v rámci jediné sekundy, druhé volání bude promeškáno, neboť kód procedury právě čeká v příkazu pause.

3. INTEGRACE OCL S JINÝMI TECHNOLOGIEMI

Interoperabilita (schopnost spolupráce s jinými systémy) je klíčovou vlastností požadovanou zákazníky z průmyslové a obchodní sféry. Interoperabilita chrání investice zákazníků, redukuje vázanost na jediného dodavatele a umožňuje implementaci rozsáhlých informačních systémů zahrnujících ekonomiku i výrobu.

Interoperabilita je široký pojem – spolupráce informačních systémů se může odehrávat na řadě úrovní a může zahrnovat prostou výměnu dat bez synchronizace s reálným časem i vzdálené volání procedur v reálném čase. Nároky na efektivitu výměny dat a časové odezvy se v jednotlivých případech velmi liší.

Schopnost spolupráce se vstupně/výstupními jednotkami či průmyslovými automaty řady výrobců je nezbytný požadavek pro všechny vizualizační a řídicí systémy. Protože v řadě aplikací procházejí přes ovladače velké objemy dat a komunikace probíhá v reálném čase, nároky na rozhraní mezi programovým systémem a periferiemi jsou vysoké. V systému Control Web toto rozhraní implementuje vrstva abstraktních ovladačů zařízení (protokol výměny dat mezi systémem Control Web a vstupně/výstupními jednotkami byl popsán dříve).

> *Technologie ovladačů, jakožto nezávislých programových komponent s pevně definovaným procedurálním aplikačním programovým rozhraním, je v systému Control Web využita i pro zpřístupnění některých standardizovaných rozhraní (např. OPC – OLE for Process Control) a také rozhraní přímo nesouvisejících s průmyslovým hardware (např. protokol HTTP).*

Spolupráce s podnikovými ekonomickými informačními systémy zpravidla neprobíhá v reálném čase a není problém ji zajistit např. prostřednictvím sdílených databází, do nichž průmyslový systém zapisuje technologická data a ekonomický systém je vyčítá, případně opět předává informace zpět technologickému systému.

Zřejmě nejobecnější formou interoperability je schopnost spolupráce různých programových komponent v rámci jediné aplikace – komponenty mohou typicky přenášet data (vzájemně zapisovat a číst vlastnosti) a také volat metody. V aplikacích se může jednat o komponenty implementující specifické algoritmy (např. komponenty pro šifrování dat nebo pro analýzu obrazu), komponenty pro přístup k databázím, komponenty realizující komunikaci s periferiemi (např. komponenta pro ovládání webové kamery a nahrávání

obrázků), komponenty pro zobrazování specifických formátů dat (např. komponenta pro zobrazování souborů formátu Portable Document Format – PDF), komponenty tvořící prvky grafických uživatelských rozhraní či komponenty tvořící přímo jádro aplikací (např. komponenta WWW prohlížeče).

Základem programové spolupráce komponent je dodržení definovaného standardu komponentového modelu. Moderní virtuální stroje (Sun JVM, Microsoft CLR) tvoří přirozenou platformu pro vzájemnou spolupráci komponent. Komponentové systémy ale vznikaly již dříve a tyto standardy jsou dnes stále nejrozšířenější.

Ve výčtu obecných komponentových programovacích technologií stojí za zmínku komponentový model IBM SOM (System Object Model). SOM je na platformě a programovacím jazyku nezávislý objektový model (programovací jazyk pouze musí podporovat práci s adresami funkcí a jejich nepřímé volání). SOM eliminuje nevýhody tradičních objektově-orientovaných jazyků – zejména binární závislost klientů na definici tříd, problémy s dynamickou vazbou za běhu aplikace apod. Rozhraní a třídy jsou v rámci SOM definovány jazykem IDL (Interface Definition Language) a IDL překladač generuje kód pro zvolený cílový jazyk (typicky C nebo C++). SOM byl implementován na platformách IBM OS/2 a AIX a neschopnost IBM prosadit tyto platformy na trhu coby průmyslové standardy s sebou přinesl i praktický zánik této nadějné technologie.

3.1. Integrace s komponentovým modelem Microsoft COM

V prostředí Microsoft Windows se prosadil komponentový standard COM (Component Object Model). COM je binární standard postavený na pojmu „rozhraní“. Rozhraní je implementováno jako vektor ukazatelů na metody. Jakmile je jednou rozhraní zveřejněno, počet a pořadí metod i jejich signatury musí být ustáleny a není možné je dále modifikovat.

Podoba binární implementace COM rozhraní byla záměrně zvolena identická s binární podobou tabulky virtuálních metod objektů tvořených překladačem Microsoft C++. Implementovat rozhraní tak pro programátora v jazyce C++ nepředstavuje větší problém než vytvořit instanci třídy s virtuálními metodami odpovídajícími metodám implementovaného rozhraní.

COM nepracuje s mechanismy považovanými v OOP systémech za povinné (např. dědičnost) a nelze jej tedy považovat za objektově-orientovaný programovací systém (bez

ohledu na název). COM rovněž nedefinuje pojem „stav objektu“ a pojem „třída“ je výrazně omezen ve srovnání s OOP systémy – třída v COM nenese žádnou statickou typovou informaci (ačkoliv zpravidla je možné popis typů rozhraní dohledat v tzv. typové knihovně, instalované spolu s DLL implementující danou třídu). Identifikace třídy je v prostředí COM pouze prostředek jak vytvořit instanci objektu. Klienti volající metody objektu se nemohou spolehnout na existenci jakéhokoliv rozhraní, vyjma univerzálního obecného rozhraní IUnknown. Všechna rozhraní musí být získána dynamicky za běhu aplikace (právě prostřednictvím univerzálního IUnknown) bez ohledu na přítomnost či nepřítomnost jeho popisu v typové knihovně spojené s objektem.

COM je standard velice nízké úrovně. Velmi dobře řeší problémy např. s verzemi tříd a dovoluje spolupráci komponent různých výrobců implementovaných v různých programovacích jazycích. Jeho použití je ale relativně složité, implementace COM třídy a podpůrných tříd (např. ClassFactory), je náročná a aplikace používajících COM jsou obtížně laditelné. Přesto je COM nejrozšířenější a nejobecnější komponentový model a je velice hojně používán nezávislými poskytovateli software.

Oblíbenost COM komponent tkví také ve skutečnosti, že řada vývojových nástrojů vyšší úrovně skrývá celou komplexnost modelu a nabízí uživateli snadné a přímočaré použití existujícího kódu COM komponent. K podpoře nástrojů vyšší úrovně vznikly nad samotným standardem COM rozšíření, umožňující např. zabudování komponent do grafického uživatelského rozhraní hostitelské aplikace (technologie původně nazývané OLE Controls, nyní Active X), programové řízení komponent s pozdní vazbu metod za běhu aplikace a přenosem dat v typově volných záznamech (technologie COM Automation) apod.

Integrace OCL s COM komponentami (případně s Active X či COM Automation komponentami) vyžaduje implementaci částí virtuálního stroje pro podporu COM za běhu aplikace. Protože OCL je nativní programovací prostředek vývojového systému Control Web a vzájemná spolupráce mezi tzv. virtuálními přístroji (komponentami systému Control Web) a OCL kódem je implementována, byl jako most mezi světem COM komponent a OCL navržena nová třída virtuálního přístroje systému Control Web zvaná Active X kontejner. Na tento virtuální přístroj tak lze pohlížet také jako na součást virtuálního stroje jazyka OCL implementující spolupráci s COM komponentami a Active X prvky.

3.1.1. Implementace Active X kontejneru

Active X kontejner je z hlediska ostatních komponent aplikace systému Control Web zcela standardní virtuální přístroj. Základní funkcí tohoto přístroje je vytvořit prostředí pro existenci Active X komponenty v prostředí aplikace Control Web. Toto prostředí (kontejner) je tvořeno řadou instancí komponent s požadovanými rozhraními dle definice standardu COM/Active X. Popis těchto komponent a jejich rozhraní není obsahem této práce a lze jej nalézt ve firemní literatuře firmy Microsoft [6] a v další literatuře [7] a [8].

Základním parametrem virtuálního přístroje Active X kontejner je identifikace COM třídy označovaná CLSID (ClaSs IDentifier). CLSID je 128 bitové globálně unikátní číslo (GUID – Gloablly Unique IDentifier). Vzhledem k relativní obtížnosti práce s GUID pro uživatele existuje alternativní cesta identifikace COM třídy prostřednictvím textového popisu. Toto označení ale slouží jen jako identifikátor záznamu v registrační databázi operačního systému obsahující vlastní identifikaci CLSID. Oba tyto způsoby identifikace jsou v Active X přístroji podporovány.

Další vlastnosti i programové rozhraní virtuálního přístroje Active X kontejner jsou proměnné v závislosti na vlastnostech COM objektu, jehož instance je v kontejneru vytvořena:

- Přístroj může mít vizuální podobu, pokud je v něm vytvořena instance Active X prvku. Řada COM komponent, zaměřených např. na komunikaci se specifickými periferiemi, vizuální podobu nemá – pak i virtuální přístroj Active X kontejner nelze zobrazit v grafickém uživatelském rozhraní aplikace.
- Podobně jako jsou v rámci aplikace definovány počáteční hodnoty parametrů (vlastností) virtuálních přístrojů, i Active X kontejner dokáže definovat počáteční hodnoty vlastností Active X prvku. Množina těchto vlastností a jejich typy závisí třídě prvku samotného.
- Základní množina nativních OCL procedur virtuálního přístroje Active X kontejner je rozšířena o programové rozhraní vytvořeného Active X prvku. Z hlediska zbytku aplikace jsou metody COM Automation rozhraní prvku nerozlišitelné od nativních procedur přístroje. Na rozdíl od jiných tříd virtuálních přístrojů je ale množina nativních OCL procedur přístroje Active X kontejneru proměnná v závislosti na třídě použité Active X komponenty.

- Stejně jako ostatní virtuální přístroje i Active X kontejner nabízí řadu událostních OCL procedur (identifikovaných patřičnými signaturami). I tato množina není pevně dána, jako je tomu u ostatních virtuálních přístrojů systému Control Web, ale je dynamicky tvořena na podle událostního rozhraní použité COM komponenty.

Poznamenejme, že množiny datových typů definovaných v prostředí COM a v systému Control Web nejsou shodné. Z důvodů zajištění maximální bezpečnosti a kontinuálního běhu aplikací např. Control Web nepodporuje práci s ukazateli. Vlastnosti a metody COM komponenty, které s nepodporovanými datovými typy pracují, nejsou do prostředí Control Web mapovány a nelze je v aplikaci použít.

3.1.2. Rozhraní pro COM Automation

Rozhraní definovaná v prostředí COM jsou pevně typovaná a neměnná. Ačkoliv je nezbytné všechna rozhraní komponenty získat dynamicky až za běhu aplikace po vytvoření její instance, typová informace (počet, pořadí a signatury metod rozhraní) je pevně dána a klient ji musí znát – standard COM ji nedovoluje po zveřejnění měnit. Rozhraní jsou identifikována globálně unikátními identifikátory nazývanými IID (Interface IDentifier).

Pro implementaci pozdní vazby bez znalosti typové informace (což je charakteristická vlastnost volně typovaných jazyků) byl nad standardem COM vystavěno jeho rozšíření nazvané COM Automation (dříve OLE Automation).

Zjednodušeně řečeno je COM Automation postaven na dvou rozšířeních:

- Je definován výčet datových typů VARTYPE a variantní záznam VARIANT, mapující všechny podporované typy do přirozených typů daného programovacího jazyka. Součástí API je rozsáhlá podpora konverzí mezi datovými typy a strukturami VARIANT apod.
- Je definováno rozhraní IDispatch, obsahující metody pro implementaci pozdní vazby. IDispatch umožňuje navázání vazby a také zprostředkovává vlastní volání. Dynamické rozhraní COM Automation tedy není implementováno jako rozhraní podle standardu COM, ale je implementováno prostřednictvím standardního rozhraní IDispatch. Poznamenejme, že COM Automation rovněž podporuje standardní COM rozhraní, nicméně nutnost typové znalosti těchto

rozhraní při překladu (či generování kódu až za běhu aplikace) použití této varianty velmi omezuje.

Samozřejmě s sebou COM Automation přináší řadu dalších definic a standardů (např. podporu práce s dynamickými textovými řetězci apod.).

COM komponenta, která chce zpřístupnit své rozhraní dynamickým (skriptovacím) jazykům tedy musí implementovat COM rozhraní IDispatch. Jeho nedílnou součástí jsou metody pro získání typové informace o implementovaném dynamickém rozhraní. Možnost získat typovou informaci za běhu je nezbytná, aby klient komponenty mohl dynamicky navázat jednotlivé metody se správnými signaturami a aby mohl správně vytvářet parametry dynamických metod.

Této typové informace využívá Active X kontejner k vytvoření a implementaci nativních OCL procedur nabízených jiným komponentám aplikace systému Control Web. Za běhu aplikace jsou volání OCL procedur zachycována, jejich parametry jsou zapouzdřeny do struktur podle standardu COM Automation a prostřednictvím rozhraní IDispatch je volána metoda Active X komponenty. Po návratu z metody jsou výstupní argumenty a návratová hodnota zpětně propagovány do prostředí OCL.

3.1.3. Dynamická implementace událostních rozhraní COM Events

Mechanismus zpětné notifikace kontejneru o událostech, které proběhly v COM komponentě, je definován standardem nazvaným COM Events. COM Events využívá rozhraní IDispatch stejně jako je tomu u COM Automation. Událostní rozhraní ale není implementováno komponentou samotnou. Je nutné je implementovat na straně kontejneru a informovat klienta, že se na ně může napojit. Směr volání metod událostních rozhraní je tedy opačný než je tomu v případě COM Automation rozhraní.

Obecný kontejner ale nemá v době překladu žádnou typovou informaci o událostním rozhraní komponent, které v něm budou vytvářeny. Rovněž není prakticky myslitelné omezit kontejner jen na určitou předem známou množinu komponent, jejichž událostní rozhraní bude implementovat.

Tento problém je řešen dynamickou konstrukcí samotného rozhraní (tedy nikoliv dynamickou vazbou na existující rozhraní) za běhu aplikace.

Aby mohl kontejner vytvořit rozhraní za běhu aplikace, potřebuje znát jeho typovou informaci. Tuto informaci musí poskytnout komponenta sama ve své typové knihovně.

Událostní rozhraní je typově popsáno v zásadě stejně jako COM Automation rozhraní, pouze je označeno atributem jako zdrojové. Je otázkou kontejneru (v našem případě komponenty Active X kontejner), aby popis rozhraní v typové knihovně nalezl, vytvořil jeho instanci a nabídl je komponentě ke zpětnému volání.

Událostní rozhraní COM komponent v Active X kontejneru je jiným částem aplikace prezentováno jako událostní procedury virtuálního přístroje samotného, podobně jako COM Automation rozhraní komponenty je mapováno do podoby nativních OCL procedur. Jestliže například Active X prvek „Command Button“ v kontejneru nabízí metodu „Click()“ informující, že uživatel kliknul na prvek myší, pak autor aplikace má možnost implementovat OCL proceduru Click() v kontextu virtuálního přístroje Active X kontejner a tato procedura bude volána jako událostní procedura kdykoliv uživatel stiskne toto tlačítko.

3.2. OCL jako programovací jazyk HTTP serveru

Internetové technologie stále více ovládají svět informačních systémů a významnou měrou přispívají k zajištění interoperability aplikací v distribuovaném síťovém prostředí. Pojmy „program“ či „aplikace“ posouvají svůj význam – spíše než samostatný proces pracující na daném počítači zahrnuje aplikace řadu komponent pracujících na jednom či více počítačích řešících aplikační logiku, spolupracujících s databázovým strojem a prezentující data uživatelům prostřednictvím webového rozhraní.

Nejvýznamnějšími standardy v oblasti interoperability celých aplikací se zřejmě staly protokoly a formáty dat používané v rámci Internetu, především protokol HTTP pracující nad síťovou vrstvou TCP/IP a standard formátování dokumentů HTML s dalšími rozšířeními umožňujícími přesnější formátování prezentovaných dat, zpětnou vazbu uživatelů, přenosy souborů zpět na server apod.

Ke standardizaci poměrně úzké množiny protokolů přispívá i živelný a neřízený vývoj Internetu. Díky absenci jednoznačné autority regulující provoz na síti a vynucující dodržování obecných norem se prostředí Internetu stalo velice nebezpečným. V každém okamžiku v Internetu probíhá velké množství útoků (nebo pokusů o útok) mající za cíl proniknout do nechráněných počítačů a celých sítí. To vede zejména firmy, ale i ostatní organizace a soukromé osoby, k zavádění přísných bezpečnostních opatření a k izolaci lokálních sítí prostřednictvím specializovaných síťových směrovačů (firewalls). Zasílání UDP datagramů či tvorba TCP spojení je na většině IP portů blokována a zpravidla jediný

protokol, který za ochranné zdi firemních sítí pronikne, je HTTP. Jeho využití k zajištění interoperability programových systémů tak zůstává jako jediná možnost.

Tvorba distribuovaných aplikací využívajících standardních internetových technologií k propojení serverů i klientů může vyžadovat nasazení zkušených programátorských týmů se znalostmi zasahujícími do řady oborů a může být časově náročná. Angažování takových týmů nemusí být problém pro velké průmyslové podniky, zpravidla je ale nedostupné pro malé a střední firmy nebo soukromé osoby. Proto je jedním z návrhových cílů systému Control Web zpřístupnit vývoj takových aplikací (a učinit jej ekonomickým) i pro malé týmy bez speciálních znalostí a zkušeností, umožnit i jednotlivá nasazení v průmyslu, v laboratořích a školách.

Moderní WWW aplikace nemohou být tvořeny jen statickými HTML stránkami. Značná část aplikační logiky je implementována na straně serveru a OCL hraje klíčovou roli jazyka používaného v systému Control Web k implementaci aplikační logiky a ke tvorbě dynamických stránek.

3.2.1. HTTP server v systému Control Web

Integrace HTTP serveru využívá komponentová architektury celého systému. Je implementován jako virtuální přístroj (podobně jako např. výše zmíněný Active X kontejner). HTTP server se svou funkčností poněkud vymyká ostatním virtuálním přístrojům, specializovaným na implementaci aplikační logicky a uživatelského rozhraní průmyslových řídicích a vizualizačních aplikací. Pro virtuální přístroje je typické, že vykonávají svou činnost během tzv. aktivací. Příčin aktivace je řada – od periodické aktivace v daných časových okamžicích (Control Web je systém reálného času), přes explicitní aktivaci plynoucí z aplikační logiky (aktivace způsobená činností jiného virtuálního přístroje), až po aktivace způsobené změnami hodnot datových elementů nebo uživatelskou událostí. HTTP server je ale aktivován po přijetí požadavku klienta. Protože tyto požadavky jsou v principu asynchronní a jejich zpracování je implementováno v řadě prováděcích toků, je nutné zajistit synchronizaci s ostatními virtuálními přístroji a se systémem komunikace s průmyslovým hardware, což jsou úkoly prováděné v rámci základního prováděcího toku celého systému.

Z komponentového návrh celého systému plyne jedna důležitá skutečnost – HTTP server je v aplikaci přítomen (jeho DLL je mapována do adresového prostoru procesu), jen pokud se k tomu návrhář aplikace explicitně rozhodne. Neexistuje tedy žádný „předdefinovaná“

WWW podoba aplikace, HTTP server nikdy nezobrazuje přímo data z technologie a nenastavuje přímo datové elementy. Vždy je nutno explicitně zadat přemapování mezi světem aplikace a vnějším světem. Neexistuje tedy možnost zobrazit nebo nastavit hodnoty aplikace z WWW klienta nebo i speciálně napsaným programem, pokud je návrhář aplikace pomocí HTTP přístroje nepřemapuje a tím nezveřejní.

Integrace HTTP serveru do systému Control Web dovoluje evoluční přístup k tvorbě aplikací – autoři aplikací mohou využít všech znalostí a zkušeností s tvorbou klasických aplikací a WWW rozhraní aplikace doplnit později. Stejně tak úroveň WWW rozhraní může s časem narůstat od jednoduchého zobrazení základních procesních veličin po komplexní WWW aplikace.

Obecnost návrhu dovoluje libovolnou konstrukci WWW rozhraní. WWW podoba aplikace může zahrnovat prostý text či tabulky s textovou reprezentací hodnot sledovaných veličin, vhodnou po linky s nízkou přenosovou kapacitou (např. GPRS spojení). Také je možné napodobit vizuální vzhled aplikace na pracovní ploše hostitelského počítače, což je podoba samozřejmě náročnější na množství přenášených dat.

Na vlastní server, vyhovující normě protokolu HTTP [9], navazují další technologie:

- HTTP server je integrován se systémem komunikace s průmyslovým hardware (čtení a zápis technologických veličin prostřednictvím ovladačů).
- Do dokumentu lze dynamicky vygenerovat jakoukoliv část textu jako výsledek vyhodnocení výrazu nebo text generovaný libovolnou procedurou.
- Celý dokument může být vytvořen kompletně dynamicky kódem procedury, případně může kód procedury přesměrovat požadavek na jiný soubor apod.
- Požádá-li klient o obrázek přemapovaný jako vzhled přístroje, server dynamicky vrátí klientovi okamžitou grafickou podobu přístroje, aniž by tato podoba existovala jako soubor.
- Požádá-li klient o specifikovanou stránku, přístroj HTTP serveru dokáže aktivovat libovolné přístroje v běžící aplikaci ještě před vygenerováním a odesláním dokumentu.
- Data vrácená z HTML formulářů lze promítnout zpět do řízeného procesu. Technologický proces lze prostřednictvím HTTP protokolu nejen vizualizovat, ale i řídit.

Prostřednictvím HTML formulářů lze nejen nastavovat hodnoty datových elementů, ale i aktivovat libovolné jiné přístroje v aplikaci či volat procedury.

3.2.2. Dynamické generování dokumentů

Jedním ze základních rozlišovacích znaků systému Control Web a jiných prostředí pro rychlý vývoj aplikací (např. Visual Basic) je možnost vytvořit funkční aplikaci bez programování (programováním zde rozumíme zápis algoritmu v daném programovacím jazyce). Stejně tak je možné v systému Control Web vytvořit i dynamické WWW stránky bez programování (tedy bez použití OCL). Pokud je aplikace složitější (např. spolupracuje s databází apod.), je použití OCL nezbytné. Nahrazování klíčů (textových podřetězců) na základě vyhodnocení výrazů je dostatečné pro celou řadu případů, existují ale situace, kdy tento způsob není použitelný. Zejména pokud délka nově generovaného textu není dopředu známa a nelze ji popsat přímo v HTML dokumentu. Typickým příkladem může být generování tabulky s předem neznámým počtem řádků, např. při konverzi výsledků databázového dotazu do podoby HTML dokumentu.

K algoritmickému generování části HTML stránky slouží mechanismus podobný přemapovávání klíčů výsledky výrazů. Je také založen na definici seznamu klíčů vyhledávaných v HTML stránkách v okamžiku odesílání dokumentu klientovi. Liší se jen způsobem tvorby řetězce, kterým je nalezený klíč v HTML dokumentu nahrazen. Řetězec není získáván jako textová podoba výsledku vyhodnocení výrazu, ale musí jej vytvořit kód OCL procedury k tomuto účelu vyvolané. Zápis přemapování může vypadat např. takto:

```
calls
  item
    id = '_put_table_';
    call = GenerateTable( NumberOfLines );
  end_item;
  item
    id = '_put_list_';
    call = GenerateList( NumberOfLines );
  end_item;
end_calls;
```

Volané procedury mají ke generování textu, jež nahradí odpovídající řetězec, k dispozici tři nativní procedury přístroje HTTP serveru:

- PutText(s : string) – volání procedury umístí do HTML dokumentu řetězec „s“.
- PutCRLF() – volání procedury umístí do HTML dokumentu znaky nového řádku (CR a LF). Dělení řádků má z hlediska HTML stejný význam jako mezera a je smysluplné, jen pokud bude výsledný text prohlížen ve své zdrojové podobě.

Z hlediska WWW prohlížeče jsou ale znaky CR a LF nepodstatné a jejich generování do dokumentu tak může být zbytečné zdržení a zvětšení objemu přenášených dat.

- PutFile(FileName : string) – volání procedury umístí do dokumentu obsah souboru, jehož cesta je uvedena v parametru FileName. Z důvodů bezpečnosti je maximální délka souboru omezena na 256 KB. Pokud je soubor delší, nebude do textu vložen a volání bude ignorováno.

V rámci těla procedury je samozřejmě možné (a většinou nutné) volat proceduru PutText vícekrát. Každé volání připojí obsah řetězcového parametru k bloku textu, který je před vyvoláním generující procedury automaticky vyprázdněn. Text se považuje za ukončený v okamžiku ukončení generující procedury.

Procedura GeterateTable z předchozího příkladu tak může vypadat například následovně:

```
procedure GenerateTable( lines : longint );
var
  i : longint;
begin
  PutText( '<table border = "1">' );
  for i = 1 to lines do
    PutText( '<tr><td> line </td><td> ' + str( i, 10 ) +
             ' </td></tr>' );
    PutCRLF();
  end; (* for *)
  PutText( '</table> ' );
end_procedure;
```

Při práci s procedurami volanými v rámci generování HTML stránek platí následující pravidla:

- Nativní procedury PutText, PutFile a PutCRLF jsou funkční jen pokud přístroj HTTP serveru vyvolal nějakou proceduru přemapovanou v rámci sekcí „calls“ nebo „pages“ a jen u toho přístroje HTTP serveru, který volání právě provádí. Tento přístroj použije předaných řetězců k nahrazení klíče nalezeného v HTML dokumentu. Volání těchto procedur v jiných situacích nebo jinému přístroji není chybou, ale nemá žádný význam.

- Procedury vyvolané v rámci generování HTML stránek nemusí sloužit ke generování HTML dokumentu. Pokud taková procedura nebude volat PutText, PutFile nebo PutCRLF, nalezený klíč bude z HTML dokumentu odstraněn (nahrazen prázdným řetězcem) a volání procedur lze použít ke konstrukci aplikační logiky.

- Parametry volaných procedur mohou obsahovat libovolné výrazy, které jsou v rámci volání vyčísleny. Pokud výraz obsahuje kanál, systém provede jeho změření.
- První výskyt zpožďovací instrukce yield, pause nebo wait v rámci vyvolané procedury ukončí generování HTML dokumentu. Ačkoliv kód procedury doběhne později, na podobu výsledného HTML dokumentu již nebude mít vliv. Jelikož jsou tyto procedury volány jako reakce na požadavky uživatelů (HTTP requests), může při použití zpožďovacích instrukcí docházet k rekurzivnímu volání – použití těchto instrukcí se proto nedoporučuje.

Má-li být nějaká část HTML stránek dynamicky generována ať již pomocí výrazu nebo procedury, musí být minimálně kostra stránky existovat v podobě souboru na disku. Někdy může být výhodnější, když se systém obejde zcela bez souboru. Ke generování HTML stránek zcela dynamicky, tedy kódem procedur bez nutnosti existence kostry HTML stránky jako souboru, slouží zápis volání procedur v sekci „pages“. V tomto případě není soubor načítán z disku a prohledáván na výskyt klíčů, ale celá stránka je tvořena dynamicky. Identifikaci netvoří textový klíč, ale celá URL předávaná v hlavičce protokolu HTTP.

```
pages
  item
    path = '/';
    call = GenerateIndex();
  end_item;
  item
    path = '/page1.htm';
    call = GeneratePage1();
  end_item;
end_pages;
```

Pravidla pro volání procedur za účelem generování HTML stránky jsou popsána dříve. Nicméně je nutné mít na paměti, že v případě volání ze sekce „pages“ procedura negeneruje pouze fragment stránky, ale celou stránku a ta musí odpovídat pravidlům pro tvorbu HTML dokumentů. Procedura GenerateIndex() z předchozího příkladu tak může vypadat například následovně:

```
procedure GenerateIndex();
begin
  PutText( '<html><head><title>Dynamická stránka' +
           '</title></head>' );
  PutText( '<body><h1>Stránka generovaná dynamicky ' +
           'přístrojem HTTPD</h1>' );
  PutText( '<p>Přechod na další <a href="/page1.htm">' +
           'stránku</a></p></body></html>' );
end_procedure;
```

4. ZÁVĚR

Nástroje rychlého vývoje aplikací prošly evolučním vývojem a postupně se stále více prosazují v prostředí průmyslových zakázkových systémů, neboť prokázaly schopnost řešit praktické problémy lidí. Jejich další vývoj je opět založen na evolučním principu – prosazují se standardy, které skutečně fungují a které jsou používány. Tomuto trendu se přes tradiční konzervatizmus nemůže vyhnout ani obor průmyslové automatizace a rostoucí nasazení počítačů, databázových aplikací a aplikací využívající intranetových technologií v průmyslové automatizaci to potvrzuje. Úkolem lidí vyvíjející nástroje pro tvorbu těchto aplikací je umožnit odborníkům jiných profesí tyto standardy úspěšně a jednoduše implementovat.

Aplikace vytvořené v prostředí Control Web hojně využívající OCL spolehlivě pracují na řadě míst. Systémy slouží přesně tam, kam byl vývoj směřován – řízení a monitorování technologických procesů a výrobních linek, zpřístupnění technologie řadě klientů přes WWW prohlížeč, mosty mezi databázemi podnikových informačních systémů, GSM sítěmi a světem WWW apod.

Přitom aplikace dokázali implementovat lidé, jejichž specializací je řízení průmyslových procesů, procesní inženýrství, automatizace budov apod., a nikoliv informační technologie a programování.

5. PŘÍLOHA A: SYNTAXE JAZYKA OCL

V zápisu syntaxe jazyka OCL jsou použity tyto typografické konvence:

- nonterminály jsou psány velkými písmeny: PROCEDURE
- terminály (klíčová slova, delimitery, …) jsou tučně: **while ( )**
- hranaté závorky [ ] označují žádné nebo nejvýše jedno opakování
- složené závorky {} označují žádné nebo více opakování
- svislá čára | označuje alternativní výskyt
- šipka → odděluje levou a pravou stranu pravidla

```
PROCEDURE      → procedure ( PARAMETER_LIST ) ;
                 { DECLARATION }
                 begin
                   BLOCK
                 end_procedure ;

PARAMETER_LIST → [ PARAMETER ] { ; PARAMETER }

PARAMETER      → [ [ var ] Identifier { , Identifier } : [ array of ]
TYPE ]

DECLARATION    → label LABEL_LIST ;
               | const CONST_LIST ;
               | var VAR_LIST ;
               | static VAR_LIST ;

LABEL_LIST     → [ Identifier ] { , Identifier }

CONST_LIST     → [ Identifier = CONST_EXPR ]
                 { ; Identifier = CONST_EXPR }

VAR_LIST       → [ VARIABLE ] { ; VARIABLE }

VARIABLE       → Identifier : TYPE [ , CONST_EXPR ]
               | Identifier : array [ CONST_EXPR .. CONST_EXPR ] of TYPE

TYPE           → boolean
               | shortint
               | integer
               | longint
               | shortcard
               | cardinal
```

```
                | longcard
                | shortreal
                | real
                | string
                | data

BLOCK           → [ STATEMENT ] { ; STATEMENT }

STATEMENT       → if EXPR then
                    BLOCK
                  { elsif EXPR then
                    BLOCK }
                  [ else
                    BLOCK ]
                  end
                | loop
                    BLOCK
                  end
                | while EXPR do
                    BLOCK
                  end
                | repeat
                    BLOCK
                  until EXPR
                | for Identifier = EXPR to EXPR [ by CONST_EXPR ] do
                    BLOCK
                  end
                | switch EXPR of
                  { case CONST_EXPR { , CONST_EXPR } :
                    BLOCK }
                  [ else
                    BLOCK ]
                  end
                | goto Identifier
                | return [ EXPR ]
                | Identifier : [ STATEMENT ]
                | Identifier := EXPR
                | move Identifier [ SIMPLEEXPR ],
                  Identifier [ SIMPLEEXPR ], SIMPLEEXPR
                | CALL

STATEMENT       → exit
                | continue

CALL            → [ OBJECT_NAME . ] MethodName ( PARAM_LIST )

OBJECT_NAME     → self | Identifier
```

```
PARAM_LIST      → [ PARAM ] { , PARAM }

PARAM           → Identifier | EXPR

EXPR            → SIMPLEEXPR [ RELOP SIMPLEEXPR ]

RELOP           → = | # | <> | < | > | <= | >=

SIMPLEEXPR      → TERM { ADDOP TERM }

ADDOP           → + | - | or | xor | | | ^

TERM            → FACTOR { MULOP FACTOR }

MULOP           → * | / | % | and | &

FACTOR          → ( EXPR )
                | NEGOP FACTOR
                | SIGNOP FACTOR
                | BUILTIN
                | CALL
                | Identifier
                | Identifier [ SIMPLEEXPR ]

NEGOP           → not | ~

SIGNOP          → + | -

BUILTIN         → size ( SIMPLEEXPR )
                | iif ( EEXPR, EXPR, EXPR )
                | loindex ( Identifier )
                | hiindex ( Identifier )
                | rand ()
                | sqrt ( SIMPLEEXPR )
                | sin ( SIMPLEEXPR )
                | cos ( SIMPLEEXPR )
                | tan ( SIMPLEEXPR )
                | asin ( SIMPLEEXPR )
                | acos ( SIMPLEEXPR )
                | atan ( SIMPLEEXPR )
                | sinh ( SIMPLEEXPR )
                | cosh ( SIMPLEEXPR )
                | tanh ( SIMPLEEXPR )
                | log ( SIMPLEEXPR )
                | ln ( SIMPLEEXPR )
                | exp ( SIMPLEEXPR )
```

```
| abs ( SIMPLEEXPR )
| sgn ( SIMPLEEXPR )
| trunc ( SIMPLEEXPR )
| floor ( SIMPLEEXPR )
| ceil ( SIMPLEEXPR )
| frac ( SIMPLEEXPR )
| round ( SIMPLEEXPR )
| pow ( SIMPLEEXPR, SIMPLEEXPR )
| atan2 ( SIMPLEEXPR, SIMPLEEXPR )
| min2 ( SIMPLEEXPR, SIMPLEEXPR )
| max2 ( SIMPLEEXPR, SIMPLEEXPR )
| shr ( SIMPLEEXPR, SIMPLEEXPR )
| shl ( SIMPLEEXPR, SIMPLEEXPR )
| round2 ( SIMPLEEXPR, SIMPLEEXPR )
| val ( SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR )
| length ( SIMPLEEXPR )
| caps ( SIMPLEEXPR )
| lows ( SIMPLEEXPR )
| pos ( SIMPLEEXPR, SIMPLEEXPR )
| concat ( SIMPLEEXPR, SIMPLEEXPR )
| insert ( SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR )
| item ( SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR )
| slice ( SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR )
| delete ( SIMPLEEXPR, SIMPLEEXPR, SIMPLEEXPR,
          SIMPLEEXPR )
| str ( SIMPLEEXPR, SIMPLEEXPR )
```

Příkazy „exit“ a „continue“ jsou povoleny jen uvnitř těla příkazu cyklu „loop .. end“.

6. PŘÍLOHA B: INSTRUKCE VIRTUÁLNÍHO STROJE

Instrukční soubor virtuálního stroje obsahuje tyto instrukce:

```
Jump              // nepodmíněný skok
  Offset          // cíl skoku
JumpTrue          // skok při splnění podmínky
  Offset          // cíl skoku
  PExpr           // podmínkový výraz
JumpFalse         // skok při nesplnění podmínky
  Offset          // cíl skoku
  Pexpr           // podmínkový výraz
JumpTable         // skok podle hodnot v tabulce
  NCases          // počet položek
  PCases          // pole položek (obsahuje hodnoty a cíle skoků)
  Offset          // cíl skoku, pokud nevyhovuje žádná položka
  PExpr           // výraz podle něhož se skáče
SetVar            // přiřazení proměnné
  PVar            // proměnná
  PExpr           // výraz který má být přiřazen
SetArray          // přiřazení prvku pole
  PVar            // proměnná typu pole
  PExpr           // výraz který má být přiřazen
  PIndexExpr      // indexový výraz
Move              // přesun bloků polí
  PVarFrom        // zdrojové pole
  PIndexFrom      // indexový výraz zdrojového pole
  PVarTo          // cílové pole
  PIndexTo        // indexový výraz cílového pole
  PNum            // výraz počtu
Call              // volání metody
  ObjectHandle    // odkaz na objekt
  MethodId        // handle metody
  PParameters     // pole parametrů (výrazů či polí)
  NumParameters   // počet parametrů
Return            // návrat z procedury
  PExpr           // výraz návratové hodnoty (může být NIL)
```

Jako příklad sémantiky řídicích příkazů jazyka OCL je uveden překlad řady procedur v jazyce OCL. Tento kód nedává žádný smysl, jen definuje funkčnost použitých řídicích příkazů.

```
procedure AssignProc();
begin
  a = 1;
  b[0] = 2;
end_procedure;
```

```
00000000 SetVar     a, 1
00000010 SetArray   b[ 0 ], 2
00000024 End
```

```
procedure IfProc();
begin
  if a = 0 then
    Body();
  elsif a = 1 then
    Body();
  else
    Body();
  end;
end_procedure;
```

```
00000000 JumpFalse  a = 0, 00000030
0000000C Call       Body()
00000028 Jump       0000007C
00000030 JumpFalse  a = 1, 00000060
0000003C Call       Body()
00000058 Jump       0000007C
00000060 Call       Body()
0000007C End
```

```
procedure LoopProc();
  loop
    Body();
    continue;
    Body();
    exit;
    Body();
  end;
end_procedure;
```

```
00000000 Call       Body()
0000001C Jump       00000000
00000024 Call       Body()
00000040 Jump       0000006C
00000048 Call       Body()
00000064 Jump       00000000
0000006C End
```

```
procedure WhileProc();
begin
  a = 0;
  while a < 10 do
    a = a + 1;
  end;
end_procedure;
```

```
00000000 SetVar     a, 0
00000010 JumpFalse  a < 10, 00000034
0000001C SetVar     a, a + 1
0000002C Jump       00000010
00000034 End
```

```
procedure RepeatProc();
begin
  a = 0;
  repeat
    a = a + 1;
  until a = 10;
end_procedure;
```

```
00000000 SetVar     a, 0
00000010 SetVar     a, a + 1
00000020 JumpFalse  a = 10, 00000010
0000002C End
```

```
procedure ForProc();
begin
  for a = 1 to 10 by 2 do
    Body();
  end;
```

```
00000000 SetVar     a, 1
00000010 JumpTrue   a > 10, 0000006C
0000001C Call       Body()
00000038 JumpTrue   a >= 10, 0000006C
00000054 SetVar     a, a + 2
```

end_procedure;	00000064	Jump	0000001C
	0000006C	End	
procedure SwitchProc();	00000000	JumpTable	a
begin			0, 0000001C
switch a of			1, 00000040
case 0:			2, 00000040
Body();			else 00000064
case 1,2:	0000001C	Call	Body()
Body();	00000038	Jump	00000080
else	00000040	Call	Body()
Body();	0000005C	Jump	00000080
end;	00000064	Call	Body()
end_procedure;	00000080	End	
procedure GotoProc();	00000000	SetVar	a, 0
label	00000010	SetVar	a, a + 1
l1;	00000020	JumpFalse	a < 10, 00000034
begin	0000002C	Jump	00000010
a = 0;	00000034	End	
l1:			
a = a + 1;			
if a < 10 then			
goto l1;			
end;			
end_procedure;			

7. LITERATURA

[1] Kolektiv autorů: Control Web 2000, Computer Press, 1999 (ISBN 80-7226-258-0)

[2] Kolektiv autorů: Control Web 5, uživatelská příručka, referenční příručka, Moravské přístroje a.s., 2004

[3] Niklaus Wirth: Compiler Contruction, Addison Wesley Longman Limited, 1996 (ISBN 0-201-40353-6)

[4] Common Language Infrastructure (CLI), Partition I: Concepts and Architecture, Microsoft Corporation, 2002

[5] Guido van Rossum: Python Reference Manual, Release 2.4.2, September 2005 (http://www.python.org/doc/ref/ref.html)

[6] The Component Object Model, Win32 Platform SDK Documentation, Microsoft Corporation

[7] Kraig Brockschmidt: Inside OLE 2, Microsoft Press, 1994 (ISBN 1-55615-618-9)

[8] Adam Denning: Active X Controls Inside Out, Second Edition, Microsoft Press, 1997 (ISBN 1-57231-350-1)

[9] RFC 2616 – Hypertext Transfer Protocol – HTTP/1.1

[10] RFC 1867 – Form-based File Upload in HTML

[11] Charles André: A Synchronous Approach to Reactive System Design (http://www.i3s.unice.fr/)

[12] Nicolas Halbwachs, Pascal Raymmond: A tutorial of Lustre, January 2002 (http://www-verimag.imag.fr/)

[13] P. Caspi, N. Halbwachs, D. Pilaud, P. Raymond: The Synchronous Data Flow Programming Language Lustre, Proceeding of the IEEE, 79(9), 1991, p. 1305-1320

[14] F. Boussinot, R. De Simone: The Esterel Language, Proceeding of the IEEE, 79(9), 1991, p. 1293-1304

[15] G. Berry: The Foundations of Esterel, in „Proof, Language, and Interaction: Essay in Honor of Robin Milner“, Editors: G Plotkin, C. Stirling, and M. Tofte, MIT Press, 2000

[16] G. Berry: The Esterel v5 Language Primer, Version 5.21 release 2.0, April 1999 (http://www.inria.fr/meije/esterel)

[17] Robert W. Sebesta: Concepts of Programming Languages, Addison Wesley Longman Limited, 6th edition, 2003 (ISBN 978-0321193629)

[18] David A. Watt: Programming Language Syntax and Semantics, Prentice Hall International (UK) Ltd, 1991

[19] Kenneth Slonneger, Barry L. Kurtz: Formal Syntax and Semantics of Programming Languages, Addison-Wesley Publishing Company, 1995 (ISBN 0-201-65697-3)

8. ŽIVOTOPIS AUTORA

Osobní údaje:	Jméno:	Pavel Cagaš
	Datum narození:	5. února 1966
	Rodinný stav:	ženatý
Vzdělání:	1981-1984:	gymnázium Zlín
	1984-1989:	FE VUT Brno
	1989:	státní závěrečná zkouška
Ocenění během studia:	1986:	Bronzový odznak VUT
	1987:	Stříbrný odznak VUT
	1988:	Zlatý odznak VUT
	1989:	Cena rektora VUT
Zaměstnání:	1989:	FE VUT Brno
	1990:	FT VUT Brno (Zlín)
	1990 – dosud:	Moravské přístroje a.s., Zlín
		vedoucí vývojového oddělení

Realizované projekty:

- 1990 – dosud: vedoucí týmu vývoje software z oblasti průmyslové automatizace
- 1990: Sada antivirových programů SafetyLab s dynamickým hashovacím algoritmem pro rychlé vyhledávání podřetězců (vzorků virů).
- 1991: Grafický systém nezávislý na zařízení a uživatelské rozhraní pro počítače PC, událostmi řízené grafické uživatelské rozhraní (GUI) Views.
- 1992: Port GUI do chráněného režimu (Protected Mode) procesorů Intel, adaptace TS DOS Extenderu, správa virtuální paměti, kooperativní řízení procesů, DLL, systém správy oken a sada ovládacích prvků.
- 1993: Systém uživatelského rozhraní orientovaného na dokumenty, koncept obecného dokumentového kontejneru, vzájemné zabudovávání dokumentů a jejich aktivace (editování) změnami kontextu aplikací.

- 1993 - dosud: Vedení návrhu a implementace systémů pro řízení a monitorování průmyslových procesů Control Panel a Control Web (tyto systémy byly v letech 1996 a 2000 oceněny cenou Křišťálový disk na výstavách Invex Comuter, jsou lokalizovány do angličtiny, němčiny a japonštiny a distribuovány na Slovensku, v Německu a Japonsku), implementace těchto pod-projektů:
- Komponentový systému s dynamickým zaváděním komponent. Dynamické rozhraní komponent s run-time detekcí rozhraní s pozdní vazbou, používané skriptovacím jazykem.
- Skriptový jazyk OCL vycházející ze syntaxe modulárních jazyků (Pascal, Modula-2), překladač do jazyka virtuálního stroje a návrh a implementace tohoto stroje.
- Programátorský editor s pokročilými funkcemi (undo-redo, zvýraznění textu dle syntaxe, apod.).
- Některé virtuální přístroje systémů Control Panel a Control Web.
- Adaptace systému pro práci na embedded PC s limitovanými zdroji (v systémech DOS zaváděných z CF paměťových karet).
- Port aplikačního rozhraní do prostředí Win32. Abstraktní objektová vrstva Views nad Win3 API, adaptace sémantiky šíření událostí Win32 na sémantiku Views.
- Windows NT kernel-driver pro komunikaci s I/O kartami.
- Implementace obecného COM/Active X kontejneru, zabudování Active X komponent do prostředí Control Web.
- Přemostění mezi OCL a ActiveX Scripting (vazba na skriptovací rozhraní COM komponent). Volání COM metod z OCL, volání OCL událostních metod z COM komponent (dynamická implementace Event Source Interface).
- Implementace HTTP serveru s dynamickou tvorbou stránek v jazyce OCL (server-side scripting), dynamickým generováním obrázků, vazba serveru na zbytek systému Control Web.
- Práce na adaptaci systému Control Web pro práci se znakovou sadou UNICODE, jeho integrace s podsystémem pro vstup nelatinských jazyků IME v prostředí Windows 2000/XP.

- Implementace klientské i serverové části standardního průmyslového rozhraní OPC (OLE for Process Control) Data Access, založeného na komponentovém modelu COM.
- Práce na přenosu aplikačního rozhraní systému Control Web do prostředí Windows CE
- Návrh a implementaci firmware USB zařízení (USB device) pro průmyslové modulární vstupně/výstupní jednotky DataLab, implementace WDM ovladače (USB function driver) pro Windows 2000 a XP.
- Kompletní návrh elektroniky, mechaniky i programové podpory (firmware, systémové ovladače pro OS Windows, uživatelské aplikace) řady integrujících slow-scan CCD kamer pro podmínky s extrémně nízkou úrovní osvětlení (autoemisní elektronová mikroskopie, astronomie, …) s USB rozhraním.
- Návrh a implementace programu SIMS (Simple Image Manipulation System) pro řízení astronomických CCD kamer, manuální i automatizované snímání, kalibraci snímků apod. Program SIMS obsahuje řadu pokročilých funkcí pro manipulaci s obrazem (sub-pixelové sčítání, filtry, detekce a identifikace hvězd s eliminací gradientů, artefaktů apod.). Splu s kamerami G2CCD je používán mnohými amatérskými i profesionálními astronomickými pracovišti i výzkumnými a vzdělávacími institucemi (Ústav fyzikální chemie Jaroslava Heyrovského, VŠCHT Praha, apod.).

Publikace:

- Cagaš P.: Průmyslová automatizace ve věku internetu, Sdělovací technika 12/1998
- Cagaš P.: Control Web a internetové technologie, Automatizace 3/1999
- Kolektiv autorů: Control Web 2000, Computer Press 1999, (ISBN 80-7226-258-0)
- Cagaš P.: Bohatost aplikace proti jejímu dosahu, Automa 1/2001
- Cagaš P.: Control Web v kritických průmyslových aplikacích, Automa 3/2001
- Cagaš P.: Jedna velikost nepadne všem – operační systémy pro zabudované aplikace. Automa 11/2001
- Kolektiv autorů: Control Web 5 – uživatelská příručka, Moravské přístroje a.s., 2004

- Cagaš P.: Principy dynamických webových aplikací, Automa 6/2004
- Cagas P., Dvorak R. and Stehlik P.: Remote control of equipment for thermal treatment of waste gases with the aid of software system "Control Web", 8th Conference on Process Integration, Modeling and Optimization for Energy Saving and Pollution Reduction PRES 2005, Proceedings on CD ROM, Giardini Naxos, Italy (15 - 18 May, 2005)