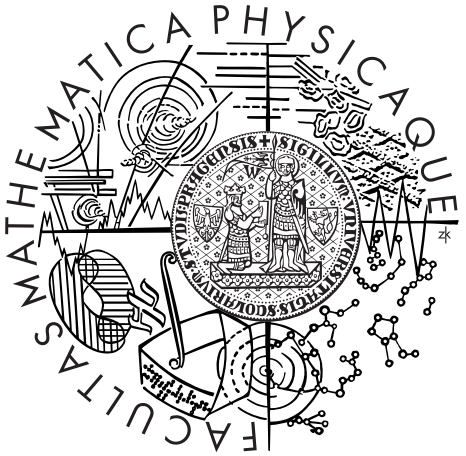
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Vlastimil Dort

Ověřování integritních omezení v C# pomocí Code Contracts

Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jakub Malý, Ph.D.

Studijní program: Informatika

Studijní obor: Obecná informatika

Praha 2014

Děkuji vedoucímu RNDr. Jakubu Malému, Ph.D. za užitečné připomínky a vstřícný přístup po celou dobu řešení práce.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Název práce: Ověřování integritních omezení v C# pomocí Code Contracts

Autor: Vlastimil Dort

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jakub Malý, Ph.D., Katedra softwarového inženýrství

Abstrakt: Jazyk OCL slouží pro specifikaci integritních omezení nad modelem jazyka UML, Code Contracts umožňují integritní omezení zapsat v programovacích jazycích pro platforum Microsoft .NET Framework. Cílem předložené práce je najít a implementovat překlad integritních omezení z jazyka OCL do programovacího jazyka C# s využitím Code Contracts. Představeny jsou možnosti technologie Code Contracts a jazyka OCL, jednotlivé jazykové konstrukce a operace ze standardní knihovny jazyka OCL jsou pak porovnány se syntakticky a sémanticky obdobnými vyjádřeními v jazyce C#. Zvolený překlad se snaží tam, kde je to možné, dodržovat sémantiku jazyka OCL. Generování zdrojových kódů v jazyce C# s Code Contracts je implementováno do programu eXolutio, ze kterého je využit editor diagramů tříd a parser jazyka OCL.

Klíčová slova: integritní omezení, code contracts, object constraint language, OCL

Title: Verifying integrity constraints in C# using Code Contracts

Author: Vlastimil Dort

Department: Department of Software Engineering

Supervisor: RNDr. Jakub Malý, Ph.D., Department of Software Engineering

Abstract: OCL serves as a language for specification of integrity constraints on UML models, Code Contract allow the integrity constraints to be expressed in programming languages targeting Microsoft .NET Framework. The goal of the thesis is to find and implement a translation of integrity constraints form the OCL language to C#, using Code Contracts. The features of Code Contracts and OCL are presented, then individual language constructs and standard library operations are compared with syntactic and semantic equivalents in C#. The chosen translation aims to match the semantics of OCL where possible. Code generation of C# sources is implemented in the eXolutio application, which provides a class diagram editor and an OCL parser.

Keywords: integrity constraits, code contracts, object constraint language, OCL

Obsah

1. Úvod

Diagramy tříd jazyka UML (Unified Modeling Language) jsou jednou z běžně používaných pomůcek při návrhu software. Modelovací nástroje s podporou UML po vytvoření modelu v podobě diagramů tříd typicky umožňují vygenerovat kód v nějakém objektově orientovaném jazyce (například C++, Java nebo C#), který může sloužit jako základ implementace programu.

Jazyk OCL (Object Constraint Language) rozšiřuje vyjadřovací schopnost jazyka UML tím, že umožňuje doplnit model o specifikaci *integritní omezení*, tedy:

- invariantů, které splňují instance tříd z modelu,
- vstupních podmínek, které musejí být splněny při volání operace,
- výstupních podmínek, které musejí být splněny při návratu z volání.

Ověřování integritních omezení ve vygenerovaném kódu umožňuje odhalit případné chyby v implementaci. Objektově orientované programovací jazyky sice často pro specifikaci integritních omezení nemají zvláštní podporu, existují však nástroje, které tuto podporu doplňují. Pro jazyk C# jsou takovým nástrojem Code Contracts, které umožňují integritní omezení zapsat v kódu a poté je ověřovat, a to jak staticky, tak za běhu programu.

Proto by bylo vhodné, podobně jako z diagramů tříd v UML generujeme třídy v jazyce C#, z integritních omezení v jazyce OCL generovat integritní omezení Code Contracts. Cílem práce tedy je:

- porovnat vyjadřovací sílu OCL a Code Contracts,
- navrhnout algoritmus překladu výrazů z OCL do C#,
- najít pro konstrukty OCL odpovídající konstrukty v Code Contracts,
- implementace navrženého překladu.

Implementační část práce je zpracována formou rozšíření existujícího programu eXolutio [XRG], který obsahuje editor diagramů tříd a editor a parser jazyka OCL.

1.1 Struktura práce

Práce začíná uvedením diagramu tříd UML s integritními omezeními OCL, který slouží jako motivační příklad a jehož úryvky jsou použity v následujících ka-

pitolách k ilustraci jednotlivých konceptů. Ve třetí kapitole jsou představeny existující projekty, které se zabývají ověřováním integritních omezení z OCL v objektově orientovaných jazycích. Ve čtvrté kapitole jsou krátce představeny elementy diagramů tříd jazyka UML, nad kterými jazyk OCL pracuje a podle kterých budou generovány kostry typů v jazyce C#. Pátá kapitola se zabývá možnostmi technologie Code Contracts, v šesté kapitole jsou postupně probrány jazykové konstrukce a standardní knihovna jazyka OCL. K jednotlivým bodům jsou uvedeny podobné nebo odpovídající konstrukce z jazyka C# nebo CodeContracts, pokud existují. V následující kapitole je popsán překlad, který byl vybrán pro implementaci v rámci programu eXolutio. Detaily této implementace jsou v osmé kapitole. Devátá kapitola obsahuje uživatelskou dokumentaci pro nově implementované funkce programu.

1.2 Použité technologie

1.2.1 Jazyk OCL

OCL (Object Constraint Language) je jazyk pro popis integritních omezení v modelech jazyka UML (Unified Modeling Language). Jazyk je vyvíjen skupinou OMG (Object Management Group), poslední verze OCL 2.4 byla vydána v roce 2014 [OCL][1].

Specifikace jazyka OCL byla předmětem kritiky pro nejednoznačnost a rozpory. Některé problémy byly v novějších verzích specifikace odstraněny, jiné jsou i v poslední verzi stále patrné[2]. Seznam chyb ve specifikaci je udržován na webových stránkách OMG [RTF].

S omezeními zapsanými v OCL se setkáme též v samotné specifikaci tohoto jazyka i v dalších částech jazyka UML, ve specifikacích UML Infrastructure, UML Superstructure [UML] a MOF.

1.2.2 Programovací jazyk C#

C# [CS] je moderní objektově orientovaný programovací jazyk vyvíjený společností Microsoft Corporation. Jazyk C# se používá především pro programování pro platformu Microsoft .NET Framework, která obsahuje běhové prostředí a rozsáhlou kolekci knihoven - FCL (Framework Class Library, [FCL]). Její jádro tvoří knihovna BCL (Base Class Library), ve které jsou mimo jiné primitivní typy jazyka C# a základní datové struktury.

[1] Předchozí verze jsou dostupné z http://www.omg.org/spec/OCL/.

[2] Specifikace například na některých místech obsahuje místo kódu integritního omezení komentář `-- TBD`, označující chybějící část.

Nové verze jazyka C# jsou vydávány společně s Microsoft .NET Framework a vývojovým prostředím Microsoft Visual Studio. V současné době nejnovější verze jazyka C# 5.0 byla vydána v roce 2012.

V této práci je C# použit jak jako cílový jazyk generování kódu, tak pro samotnou implementaci překladu.

1.2.3 Code Contracts

Code Contracts [CC, CCW], je projekt vyvíjený v rámci Microsoft Research. Sestává z knihoven a nástrojů pro platformu Microsoft .NET Framework a rozšíření pro vývojové prostředí Microsoft Visual Studio.

Code Contracts umožňují statickou i dynamickou kontrolu integritních omezení pro všechny programovací jazyky platformy .NET. Nezávislosti na programovacím jazyce se dosahuje tím, že integritní omezení jsou ve zdrojovém kódu zapsána jako běžná volání knihovních metod a až v přeloženém programu jsou tato volání detekována[3]. Poté jsou příslušným nástrojem buď staticky verifikována prostředky abstraktní interpretace, nebo jsou transformována do podoby spustitelného kódu.

Třídy technologie Code Contracts se nacházejí v jmenném prostoru `System.Diagnostics.Contracts` a jsou součástí platformy Microsoft .NET Framework od verze 4.

1.2.4 Program eXolutio

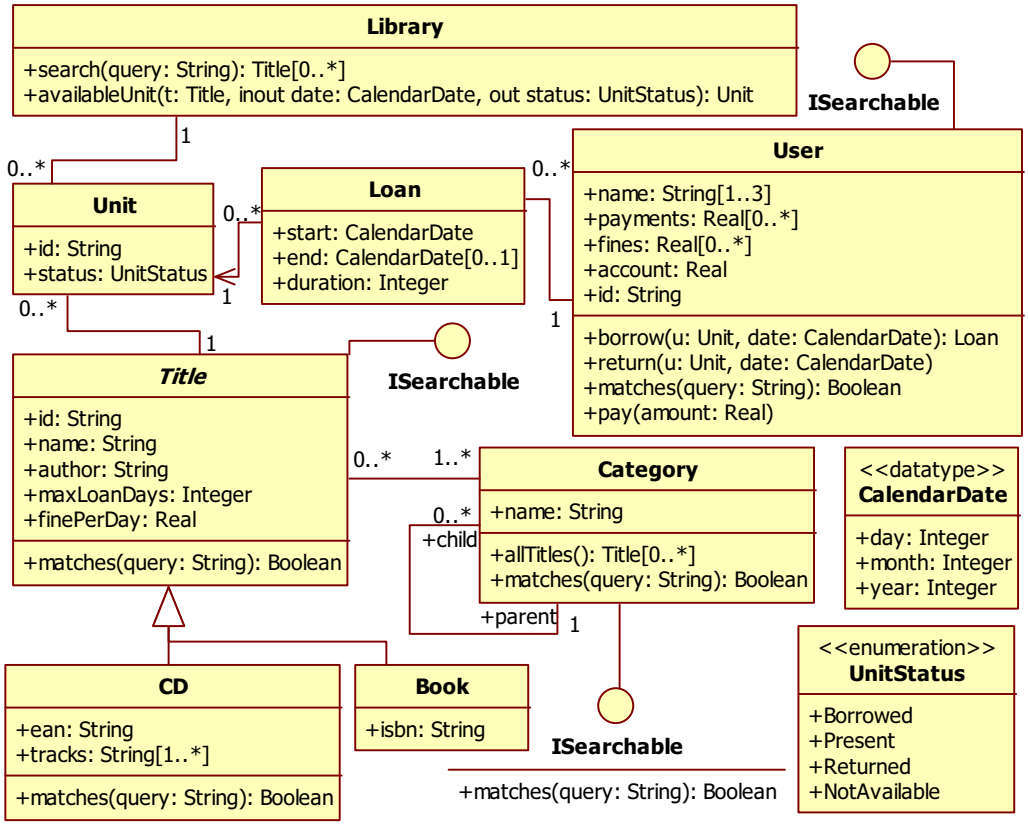
Program eXolutio [XRG] je nástroj pro vývoj schémat jazyka XML od skupiny XRG (XML Research Group) Katedry softwarového inženýrství Matematicko-fyzikální fakulty Univerzity Karlovy v Praze. Program je napsaný v jazyce C# pro platformu .NET, k vývoji je použito Microsoft Visual Studio.

[3] Tím se liší od staršího projektu Spec#, který podobný úkol řešil rozšířením syntaxe jazyka C#.

2. Motivace

2.1 Příklad diagramu tříd

Uvažujme návrh informačního systému veřejné knihovny. Postupujeme podle zásad MDA, nejprve tedy vytvoříme model v jazyce UML (obrázek 2.1).



Obrázek 2.1: Diagram tříd ukázkového modelu

V příkladu jsme modelovali:

- třídy pro jednotky a tituly s využitím dědičnosti,
- třídy pro uživatele a záznamy o jeho výpůjčkách se záznamy o pokutách za pozdní vrácení a jejich placení,
- vyhledávání titulů na pobočce, dotaz na dostupnost titulu,
- roztřídění titulů do hierarchické struktury kategorií.

Části tohoto příkladu budou využívány pro v ukázkách následujícím textu, je proto záměrně volen tak, aby na něm byly patrné možnosti jazyka UML využité

v této práci. Jeho implementaci jako projekt pro program eXolutio naleznete na přiloženém CD (příloha A).

2.2 Integritní omezení

Již se samotného modelu UML plynou některá omezení pro jeho realizaci, například že vlastnosti musejí být daného typu a dodržovat specifikovanou kardinalitu. Ze zadání ovšem plynou další podmínky, na něž samotné UML nestačí. Pro specifikaci takových podmínek je potřeba použít jiný jazyk, například OCL.

Jak ukazují výpisy 2.1 a 2.2, jazyk OCL umožňuje zadat velmi různorodé podmínky, zde konkrétně:

- Omezit číselné rozsahy položek kalendářního data, například invariant `inv: month>0 and month<=12`.
- Pomocí definování nových operací a vlastností počítat rozdíl kalendářní dat.
- Zajistit, že jednotky mají při půjčování a navracení správnou hodnotu vlastnosti `status` a je správně vyplněn záznam výpůjčky. Například `pre: u.status = UnitStatus::Present` značí vstupní podmínku, která zaručuje, že při volání operace `borrow` je jednotka u aktuálně k dispozici.
- Stanovit vztah mezi dobou výpůjček uživatele, udělenými pokutami a stavem jeho účtu. Například `derive: payments->sum()-fines->sum()` odvozuje stav uživatelova účtu od sum zaplacených částek a udělených pokut.
- Určit způsob tvoření identifikátorů pro různé entity. Například `inv: id = 'BOOK' + isbn` určuje, že identifikátor knihy je tvořen slovem „BOOK“ následovaným ISBN knihy.
- Popsat vyhledávání titulů, uživatelů a kategorií. Vyhledávání je možné přes rozhraní `ISearchable` a jeho metodu `matches`. Délka hledaného řetězce musí být kvůli vstupní podmínkce `pre: query.size()>=3` alespoň 3 znaky.
- Definovat způsob získání všech titulů z dané kategorie. Tělo metody `allTitles` třídy `Category` je definováno pomocí `body: self->closure(c|c.child).Title` jako všechny tituly patřící do dané kategorie a všech jejích podkategorií.

2.3 Realizace modelu

Díky jazykům UML a OCL jsme mohli specifikovat strukturu informačního systému a jeho vlastnosti. Pro jeho nasazení je potřeba systém implementovat ve vhodném programovacím jazyce, jímž může být C#. Jako základ takové implementace mohou posloužit zdrojové kódy automaticky vygenerované z UML modelu, které budou obsahovat definice tříd, jejich vlastnosti a deklarace metod. Pro tuto funkci je podpora ve většině modelovacích nástrojů pro UML. Příklad takto vygenerovaného kódu je ve výpisu 2.3.

Výpis 2.3 Vygenerovaná kostra třídy `Title`

```
public class Title : ISearchable {
  public virtual string Id { get; set; }
  public virtual string Name { get; set; }
  public virtual string Author { get; set; }
  ...
  [Pure]
  public virtual bool Matches(string query)
  {
    throw new NotImplementedException();
  }
}
```

Úkolem programátora pak je tyto metody implementovat. Aby bylo zajištěno, že se implementace shoduje se specifikací, chtěli bychom ověřit, že jsou v ní splněna integritní omezení, která jsme zapsali v jazyce OCL. Pokud však použijeme nástroj, který do zdrojových kódů nevkládá přeložená integritní omezení (spolu s vhodným způsobem jejich ověřování), museli bychom je přepisovat ručně, což by znamenalo vyšší časovou náročnost, možnost chyby a to zejména při dodatečných úpravách modelu.

Například se rozhodneme ověřit následující integritní omezení:

```
context Title::matches(query:String):Boolean
  post: result = (name.indexOf(query) > 0 or author.indexOf(query)>0)
```

Musíme tedy doplnit vygenerovaný kód, třeba takto:

```
public virtual bool Matches(string query) {
  Contract.Ensures(Contract.Result<bool>() ==
    (Name.IndexOf(query)>-1 && Author.IndexOf(query)>-1));
  throw new NotImplementedException();
}
```

Pokud následně dojde ke změně modelu (povolíme, aby titul měl více než jednoho autora), je nutné aktualizovat integritní omezení:

```
context Title::matches(query:String):Boolean
  post: result = (name.indexOf(query) > 0 or author->exists(indexOf(query)>0))
```

Zároveň však musíme tu samou změnu provést na našem ručně vytvořeném kódu:

```
public virtual bool Matches(string query) {
  Contract.Ensures(Contract.Result<bool>() ==
    (Name.IndexOf(query)>-1 && Contract.Exists(Author, a=> a.IndexOf(query)>-1)));
  throw new NotImplementedException();
}
```

Pokud však použitý nástroj integritní omezení generuje automaticky, stačí tuto změnu provést pouze na jednom místě, a to v modelu UML. Přegenerováním pak dostaneme kód oveřující novou verzi omezení, aniž bychom jej museli znovu zkoumat a ručně překládat.

Výpis 2.1 Skript jazyka OCL pro ukázkový model (část 1)

```
/* Kalendářní data */
context CalendarDate
  /*Omezení rozshahů*/
  inv: day>0 and day<=monthDays(month, year)
  inv: month>0 and month<=12
  inv: year>0
  /*Definice výpočtu rozdílů mezi dvěma daty*/
  static def: difference(f:CalendarDate, t:CalendarDate):Integer =
    1+t.dayInYear-f.dayInYear+Set{f.year..t.year-1}->collect(y|yearDays(y))->sum()
  def: dayInYear:Integer =
    day+Set{1..month-1}->collect(m|monthDays(m,year))->sum()
  static def: monthDays(m:Integer, y:Integer) : Integer =
    Sequence{31,if isLeapYear(y) then 29 else 28 endif,
     31,30,31,30,31,31,30,31,30,31}.at(m)
  static def: yearDays(y:Integer):Integer =
    if isLeapYear(y) then 366 else 365 endif
  static def: isLeapYear(y:Integer):Boolean =
    y.mod(4) = 0 and (y.mod(100) <> 0 or y.mod(400) = 0)

/* Půjčování */
context User::borrow(u:Unit, date:CalendarDate):Loan
  pre: u.status = UnitStatus::Present
  post: u.status = UnitStatus::Borrowed
  post: result.start = date
  post: result.Unit = u
  post: result.end->oclIsUndefined()
context User::return(u:Unit, date:CalendarDate)
  pre: Loan->exists(Unit=u and end->oclIsUndefined())
context Loan::duration : Integer
  derive: CalendarDate::difference(start, end)

/* Pokuty */
context User
  inv: payments->forAll(p|p>0)
  inv: fines->forAll(p|p>0)
  inv: fines->asBag() = Loan->collect(
    if duration.oclIsUndefined()
    then 0
    else (duration-Unit.Title.maxLoanDays).max(0)*Unit.Title.finePerDay
    endif
  )
context User::account : Real
  derive: payments->sum()-fines->sum()
context Title::maxLoanDays : Integer
  init: 30
context Title::finePerDay : Real
  init: 2.50
context User::pay(amount:Real)
  pre: amount>0
  post: payments=payments@pre->including(amount)
```

Výpis 2.2 Skript jazyka OCL pro ukázkový model (část 2)

```
/* Identifikátory */
context Book
  inv: id = 'BOOK' + isbn
  inv: isbn.characters()->select(c|c='-')->size()=4
  inv:
    let digits : Sequence(Integer) =
      isbn.characters()->select(c|c<>'-')->collect(i|
        if i='X' then 10 else i.toInteger() endif)
    in Set{1..10}->collect(i|digits->at(i)*(11-i))->sum().mod(11) = 0
context CD
  inv: id = 'CD'+ean
context User
  inv: id.indexOf(name->last().toUpperCase())=1
context Unit
  inv: id.indexOf(Title.id)=1

/* Vyhledávání */
context ISearchable::matches(query:String):Boolean
  pre: query.size()>=3
context Title::matches(query:String):Boolean
  body: name.indexOf(query) > 0 or author.indexOf(query)>0
context CD::matches(query:String):Boolean
  body: name.indexOf(query) > 0 or tracks->exists(t|t.indexOf(query)>0)
context Library::search(query:String):Set(Title)
  pre: query.size()>=3
  post: result->forAll(matches(query) and Unit->exists(Library=self))
context Library::availableUnit(t: Title, date: CalendarDate, status: UnitStatus):
   Unit
  pre: t.Unit->exists(Library=self)
  post: result.result.Library=self
  post: result.result.Title=t
  post: result.status = result.result.status
/* Kategorie */
context Category::allTitles():Set(Title)
  body: self->closure(c|c.child).Title
```

3. Podobné projekty

V této kapitole představíme 3 projekty zabývající se ověřováním integritních omezení zapsaných v OCL.

3.1 C# / OCL Compiler

Dave Arnold v roce 2004 vytvořil rozšíření překladače jazyka C# z projektu Mono zvané C# / OCL Compiler [Arnold]. Integritní omezení v jazyce OCL jsou zapsána pomocí zvláštní syntaxe a pracují přímo s typy jazyka C#. Tento program nebyl aktualizován a pracuje pouze s dnes již zastaralými verzemi C# 1.0 a OCL 2.0. Příklad ve výpisu 3.1 ukazuje syntaxi zápisu dvou jednoduchých integritních omezení.

Výpis 3.1 Příklad kódu pro C# / OCL Compiler

```
class User{
...
 OCL
 [
  "context␣User::Return(instance:Instance,␣date:CalendarDate):Loan"
  "pre:␣instance.status␣=␣UnitStatus::Present"
  "post:␣instance.status␣=␣UnitStatus::Borrowed"
 ]
 public Loan Return(Instance instance, CalendarDate date) { ... }
}
```

Způsob jejich ověření je patrný ve výpisu 3.2, jenž byl získán dekompilací výsledného binárního souboru programem ILSpy[1].

3.2 Eclipse OCL

Platforma Eclipse[2], známá především jako vývojové prostředí pro programovací jazyk Java, poskytuje různé nástroje pro modelování. Součástí sady Model Development Tools je projekt Eclipse OCL [MDT], který implementuje parser a interperter jazyka OCL nad modely UML a Ecore. Z těchto modelů lze generovat zdrojové kódy tříd v jazyce Java. Integritní omezení mohou být ve vygenerovaném kódu reprezentována dvěma způsoby, buď jako řetězcové konstanty interpretované za běhu, nebo přímo jako metody v jazyce Java [EUI]. Druhá varianta je předvedena ve výpisu 3.3. Uvedený příklad ověřuje platnost invariantu `inv: month>0 and month<=12` na hodnotovém typu `CalendarDate`. Invariant je ověřován z metody

[1] http://ilspy.net/
[2] http://www.eclipse.org/

Výpis 3.2 Ověření podmínek pomocí C# / OCL Compiler (dekompilováno)

```
public Loan Return(Instance instance, CalendarDate date)
{
  OclBoolean b = instance.status == UnitStatus.Present;
  if (!b)
  {
    throw new OCLException("Pre-Condition␣Failed.", "Client",
      "context␣User::Return(instance:Instance,␣date:CalendarDate):Loan\r\n" +
      "pre:␣instance.status␣=␣UnitStatus::Present\r\n" +
      "post:␣instance.status␣=␣UnitStatus::Borrowed",
      "'User::Return'", "Pre-Condition");
  }
  Loan result = ...
  OclBoolean b2 = instance.status == UnitStatus.Borrowed;
  if (!b2)
  {
    throw new OCLException("Post-Condition␣Failed.", "Supplier",
      "context␣User::Return(instance:Instance,␣date:CalendarDate):Loan\r\n" +
      "pre:␣instance.status␣=␣UnitStatus::Present\r\n" +
      "post:␣instance.status␣=␣UnitStatus::Borrowed",
      "'User::Return'", "Post-Condition");
  }
  return result;
}
```

`validate` třídy `LibraryValidator`, která je jednou z vygenerovaných pomocných tříd. Jak je vidět, i velmi krátké integritní omezen je přeloženo do celé strany kódu. Velkou část kódu zabírá ošetřování výjimek příkazem `try-catch` jazyka Java.

3.3 Dresden OCL

Projekt Dresden OCL [DOC] také využívá vývojového prostředí Eclipse. Z integritních omezení zapsaných v editoru skriptů jazyka OCL umí generovat dotazy v jazycích SQL nebo Java s využitím technologie AspectJ. Ukázka ve výpisu 3.4 byla vygenerována z výstupní podmínky `post: result.start = date` operace `User::borrow`. Diky AspectJ je podmínka ověřena po každém volání metody.

Výpis 3.3 Příklad ověření invariantu v Eclipse OCL

```
public boolean invmonth(final DiagnosticChain diagnostics,
 final Map<Object, Object> context) {
 /**
  * inv invmonth: month>0 and month<=12
  */
 final /*@NonNull*/ /*@NonInvalid*/ CalendarDate self = this;
 /*@Nullable*/ /*@Caught*/ Object CAUGHT_and;
 try {
  /*@NonNull*/ /*@Caught*/ Object CAUGHT_gt;
  try {
   final /*@NonNull*/ /*@Thrown*/ Object month = self.getMonth();
   final /*@NonNull*/ /*@Thrown*/
    IntegerValue BOXED_month = ValuesUtil.integerValueOf(month);
   final /*@NonNull*/ /*@Thrown*/
    Boolean gt = NumericGreaterThanOperation.INSTANCE.
    evaluate(BOXED_month, LibraryTables.INT_0);
   CAUGHT_gt = gt;
  }
  catch (Exception e) {
   CAUGHT_gt = ValuesUtil.createInvalidValue(e);
  }
  /*@NonNull*/ /*@Caught*/ Object CAUGHT_le;
  try {
   final /*@NonNull*/ /*@Thrown*/ Object month_0 = self.getMonth();
   final /*@NonNull*/ /*@Thrown*/
    IntegerValue BOXED_month_0 = ValuesUtil.integerValueOf(month_0);
   final /*@NonNull*/ /*@Thrown*/
    Boolean le = NumericLessThanEqualOperation.INSTANCE.
    evaluate(BOXED_month_0, LibraryTables.INT_12);
   CAUGHT_le = le;
  }
  catch (Exception e) {
   CAUGHT_le = ValuesUtil.createInvalidValue(e);
  }
  final /*@Nullable*/ /*@Thrown*/
   Boolean and = BooleanAndOperation.INSTANCE.evaluate(CAUGHT_gt, CAUGHT_le);
  CAUGHT_and = and;
 }
 catch (Exception e) {
  CAUGHT_and = ValuesUtil.createInvalidValue(e);
 }
 if (CAUGHT_and == ValuesUtil.TRUE_VALUE) {
  return true;
 }
 if (diagnostics != null) {
  int severity = CAUGHT_and == null ? Diagnostic.ERROR : Diagnostic.WARNING;
 String message = NLS.bind(
  EvaluatorMessages.ValidationConstraintIsNotSatisfied_ERROR_,
  new Object[]{"CalendarDate", "invmonth",
  EObjectValidator.getObjectLabel(this, context)});
 diagnostics.add(new BasicDiagnostic(severity,
  LibraryValidator.DIAGNOSTIC_SOURCE,
  LibraryValidator.CALENDAR_DATE__INVMONTH, message, new Object [] { this }));
 }
 return false;
}
```

Výpis 3.4 Příklad ověření integritního omezení v Dresden OCL

```
public privileged aspect User_PostAspect_borrow {
 /**
  * <p>Pointcut for all calls on
  * {@link library.User#borrow(library.Unit u, library.CalendarDate date)}.</p>
  */
 protected pointcut borrowCaller(library.User aClass, library.Unit u,
  library.CalendarDate date):
  call(* library.User.borrow(library.Unit, library.CalendarDate))
  && target(aClass) && args(u, date);
 /**
  * <p>Checks a postcondition for the operation
  * {@link User#borrow(, library.Unit u, library.CalendarDate date)}
  *  defined by the constraint
  * <code>context User::borrow(u: library.Unit, date: library.CalendarDate)
  *   : library.Loan
  *   post: result.start = date</code></p>
  */
 library.Loan around(library.User aClass, library.Unit u,
  library.CalendarDate date): borrowCaller(aClass, u, date) {
  library.Loan result;
  result = proceed(aClass, u, date);
  if (!result.start.equals(date)) {
   // TODO Auto-generated code executed when constraint is violated.
  String msg = "Error: Constraint 'undefined' (post: result.start = date) was" +
   " violated for Object " + aClass.toString() + "";
   throw new RuntimeException(msg);
  }
  // no else.
  return result;
 }
}
```

4. Diagramy tříd v UML

Integritní omezení jazyka OCL pracují nad modelem UML. Abychom mohli integritní omezení překládat do jazyka C#, je třeba v něm nejprve vyjádřit tento model. Na generování zdrojových kódů z modelu UML v různých jazycích včetně C# již existují mnohé v praxi používané nástroje, v programu eXolutio tato funkce ovšem není. V této kapitole je popsána velmi malá část možností diagramů tříd jazyka UML [UML, kapitola 7], která však postačuje pro tvorbu různorodých modelů a využití většiny možností jazyka OCL.

4.1 Třídy (`Class`)

Nejdůležitějším elementem v diagramech tříd jsou *třídy* (`Class`). Instancemi tříd (za běhu programu) jsou objekty. Každý objekt má jedinečnou identitu, která se nemění, mohou se však měnit jeho vlastnosti. To odpovídá referenční sémantice tříd (`class`) v jazyce C#. Třídy mohou mít předky, v programu eXolutio a jazyce C# ovšem nejvýše jednoho. Třída může být *abstraktní*, pak není možné vytvářet přímo její instance, pouze instance jejích potomků. Naopak *finální* třída potomky mít nemůže vůbec. V C# tyto třídy označíme atributy `abstract` resp. `sealed`.

```
abstract class Title { ... } // Abstraktní třída
class Book : Title { ... } // Odvozená třída
```

4.2 Rozhraní (`Interface`)

Diagramy tříd mohou též obsahovat *rozhraní* (`Interface`)[1]. Třída, která realizuje rozhraní, musí obsahovat všechny vlastnosti a operace definované v daném rozhraní. V jazyce C# může jedna třída implementovat libovolný počet rozhraní, rozhraní dokonce podporují vícenásobnou dědičnost. V této práci je ovšem dědičnost rozhraní implementována stejně jako dědičnost tříd a rozhraní tedy může mít nejvýše jednoho předka.

```
interface ISearchable { ... } // Rozhraní
class Category : ISearchable { ... } // Třída implementující rozhraní
```

[1] Rozhraní jsou ve specifikaci jazyka UML v balíku Interfaces, ostatní zde popisované entity jsou z balíku Kernel.

4.3 Hodnotové typy (`DataType`)

Hodnotové typy (v UML `DataType`) se od tříd liší tím, že jejich instance nejsou určeny identitou, ale hodnotou vlastností, které jsou neměnné.

Hodnotové typy v C# (`struct`) jsou sice předávány hodnotou, jejich datové položky se ovšem mohou měnit, což může mít překvapivé důsledky [Lippert]. Těmto problémům se lze vyhnout označením všech datových položek jako `readonly` a jejich inicializací v konstruktoru.

```
struct CalendarDate{ ... } // Hodnotový typ
```

Pod hodnotové typy se v UML řadí i primitivní a výčtové typy, stejně jako v C#.

4.4 Výčtové typy (`Enumeration`)

V jazyce UML jsou *výčtové typy* instancemi metatypu `Enumeration`. Výčtový typ specifikuje množinu možných hodnot `EnumerationLiteral`. V C# deklarujeme nový výčtový typ klíčovým slovem `enum`. Výčtové typy jsou hodnotové a jsou potomky třídy `System.Enum`. Třída `System.Enum` obsahuje užitečné metody pro práci s enumeracemi, například umožňuje získat pole všech hodnot daného typu.

```
enum UnitStatus{
  Present, Borrowed, Returned, NotAvailable
}
⇒ Enum.GetValues(typeof(Book)) // Present, Borrowed, Returned, NotAvailable
```

4.5 Vlastnosti (`Property`)

Třídy, rozhraní a hodnotové typy mohou obsahovat vlastnosti (`Property`). Mohou to být jednoduché atributy, nebo konec asociace (`AssociationEnd`)[2]. Vlastnost má typ a *kardinalitu*, určující, kolika nejméně a nejvíce hodnot může vlastnost nabývat. Pokud je horní mez kardinality větší než 1, jedná se o kolekci hodnot a podle příznaků `unique` a `ordered` se určuje, zda může kolekce obsahovat stejnou hodnotu vícekrát a zda je rozlišováno pořadí hodnot.

V jazyce C# vlastnosti sestávají ze dvou metod, *getteru* a *setteru*. Pokud má vlastnost být pouze čtena (např. v hodnotových typech), může být setter vynechán nebo znepřístupněn modifikátorem `private`. V případě, že vlastnost pouze čte a nastavuje hodnotu datové položky, může být implementována *automaticky*, tedy překladač sám vygeneruje datovou položku, getter i setter. Abstraktní třídy

[2] V programu eXolutio má každá asociace právě 2 konce. Generování vlastnosti lze potlačit tím, že *opačný* konec asociace není *navigovatelný*.

mohou obsahovat abstraktní vlastnosti. Vlastnosti mohou být *statické* (`static`), takové se nevztahují k instancím, ale k typu samotnému.

```
class Unit{
  private string id; // Datová položka vlastnosti
  public string Id {get{return id;} set{ch = id;}} // Vlastnost
  public UnitStatus Status {get; set;} // Automatická vlastnost
  ...
}
struct CalendarDate{
  private readonly int day; // Datová položka neměnné vlastnosti
  public int Day { get { return day; } } // Neměnná vlastnost
  public int Month { get; private set; } // Automatická vlastnost jen pro čtení
  ...
  public CalendarDate(int day, int month, ...){
    this.day = day; // Inicializace neměnné datové položky v konstruktoru
    this.Month = month; // Inicializace vlastnosti je pro čtení
    ...
  }
}
```

4.6 Operace (`Operation`)

Třídy, datové typy a rozhraní mohou také definovat *operace* (`Operation`). Operace mají parametry (`Parameter`), které mají jméno, typ a kardinalitu (stejně jako vlastnosti) a směr. Jeden parametr může být návratová hodnota (`return`), ostatní jsou vstupní (`in`), výstupní (`out`), nebo vstupně-výstupní (`inout`). V jazyce C# se místo operací používá označení *metody*. Návratový typ se píše před název metody, pokud metoda nevrací žádnou hodnotu, může být `void`. Výstupní a vstupně výstupní parametry jsou označeny atributy `out` a `ref`.

Operace může mít nastaven atribut `isQuery`, který značí, že operace nemění globální stav (ale může ho číst). Jen takové operace je možné použít ve výrazech jazyka OCL. To je důležité, neboť při ověřování integritních omezení pozorujeme stav programu a jeho vlastnosti, nemůžeme do něj však zasahovat. Code Contracts k tomu má atribut `System.Diagnostics.Contracts.PureAttribute`.

Operace mohou stejně jako vlastnosti také být statické nebo abstraktní.

```
class Library{
  [Pure]
  Unit AvailableUnit(Title t, ref CalendarDate date, out UnitStatus s){
    ...
  }
  ...
}
```

5. Přehled technologie Code Contracts

V této kapitole jsou stručně uvedeny možnosti technologie Code Contracts [CC] použité v jazyce C#.

5.1 Zápis integritních omezení

Integritní omezení se do kódu v jazyce C# zapisují jako volání statických metod třídy `System.Diagnostics.Contracts.Contract`. Výraz, jehož platnost integritní omezení vyjadřuje, je zadán jako běžný argument tohoto volání (typu `bool`). Díky atributům podmíněného překladu se při běžné kompilaci všechna tato volání odstraní. Jejich ověřování je možné pouze při použití nástrojů Code Contracts.

5.2 Ověřování integritních omezení

Technologie Code Contracts umožňuje ověřovat zapsaná omezení dvěma způsoby: za běhu programu, nebo staticky.

K zapnutí ověřování za běhu je potřeba program nejprve přeložit s definovaným symbolem `CONTRACTS_FULL` a na přeložený program použít nástroj `ccrewrite`. Tento nástroj provede na kódu různé úpravy, například výstupní podmínky jsou ověřovány při návratu z metody, přitom se ale zapisují na začátek metody. Například metody `Borrow` uvedené ve výpisu 5.1 jsou přepsány na ekvivalent kódu ve výpisu 5.2. Následně je možné program spustit. Pokud během běhu dojde k porušení integritního omezení, je vyvolána událost `Contract.ContractFailed`. Pokud je v programu registrována obsluha této události, může chybu ignorovat nebo požádat o vyvolání výjimky `ContractException`. Jinak dojde k zobrazení dialogového okna s chybovou hláškou nebo přechodu do ladícího programu.

Statická analýza integritních omezení se provádí nástrojem `cccheck`. Statický analyzátor se snaží dokázat, že integritní omezení v programu bude vždy splněno, ze znalosti kódu a ostatních integritních omezení. Pokud se mu to dokázat nepodaří, vypíše varování. Toto varování ale nemusí znamenat chybu v programu, v takovém případě může programátor analyzátoru pomoci přidáním dalších podmínek. Statická anylýza tedy může vyžadovat větší pozornost programátora při psaní podmínek[1].

[1] V této práci se zaměříme na ověřování za běhu, integritní omezení přeložená z jazyka OCL mohou být pro statický analyzátor příliš složitá.

Oba způsoby lze používat jednoduše díky rozšíření pro Vistual Studio[2], se kterým není třeba nástroje spouštět ručně, parametry ověřování se nastaví na kartě vlastností projektu a vše pak proběhne automaticky při sestavení projektu.

Výpis 5.1 Příklad integritních omezení v Code Contracts

```
public class User {
  IEnumerable<int> Payments { get; set; }
  [ContractInvariantMethod]
  private void Invariants(){
    Contract.Invariant(Contract.ForAll(Payments, p => p > 0));
  }
  public virtual Loan Borrow(Unit u, CalendarDate date) {
    Contract.Requires(u.Status == UnitStatus.Present);
    Contract.Ensures(u.Status == UnitStatus.Borrowed);
    return ...
  }
}
```

[2]http://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970

Výpis 5.2 Příklad integritních omezení v Code Contracts po přepsání nástrojem `ccrewrite` (dekompilováno)

```
public virtual Loan Borrow(Unit u, CalendarDate date)
{
  if (__ContractsRuntime.insideContractEvaluation <= 4) {
    try {
      __ContractsRuntime.insideContractEvaluation++;
      __ContractsRuntime.Requires(u.Status == UnitStatus.Present,
        null, "u.Status␣==␣UnitStatus.Present");
    }
    finally {
      __ContractsRuntime.insideContractEvaluation--;
    }
  }
  try{
  }
  catch (Exception arg_45_0) {
    if (arg_45_0 == null) {
      throw;
    }
  }
  Loan result = ...
  if (__ContractsRuntime.insideContractEvaluation <= 4) {
    try {
      __ContractsRuntime.insideContractEvaluation++;
      __ContractsRuntime.Ensures(u.Status == UnitStatus.Borrowed,
        null, "u.Status␣==␣UnitStatus.Borrowed");
    }
    finally {
      __ContractsRuntime.insideContractEvaluation--;
    }
  }
  this.$InvariantMethod$();
  return result;
}
[ContractInvariantMethod, CompilerGenerated]
protected override void $InvariantMethod$() {
  if (!this.$evaluatingInvariant$){
    this.$evaluatingInvariant$ = true;
    try {
      __ContractsRuntime.Invariant(Contract.ForAll<int>(
        this.<Payments>k__BackingField, (int p) => p > 0),
        null, "Contract.ForAll(Payments,␣p␣=>␣p␣>␣0)");
    }
    finally {
      this.$evaluatingInvariant$ = false;
    }
  }
}
```

5.3 Statické metody třídy `Contract`

Kontrakty metod (`Requires`, `Ensures`, `EnsuresOnThrow`)

Metoda `Contract.Requires` označuje *vstupní podmínku*, která musí být splněna při každém volání metody. *Výstupní podmínka* v `Contract.Ensures` je vyhodnocena při normálnm opuštění metody. V případě, že je vyvolána výjimka, je místo toho ověřena podmínka `Contract.EnsuresOnThrow` pro typ výjimky specifikovaný jako ge-

nerický parametr. OCL podporuje oba druhy podmínek, označuje je `pre` a `post`. Jelikož v OCL nelze rozlišovat mezi typy výjimek, je nanejvýš možné specifikovat podmínku pro všechny typy výjimek. To se ovšem nedoporučuje, neboť pak je podmínka ověřována i v případech, kdy dojde k výjimce, kterou programátor neočekával.

Kontrakty metod se dědí jak z rodičovských tříd, tak z implementovaných rozhraní. Nelze kombinovat vstupní podmínky z více metod, proto mohou být specifikovány pouze na metodách, které nemají žádné předky.

```
class User{
  void Borrow(Unit u){
    Contract.Requires(u.Status == UnitStatus.Present);
    Contract.Ensures(u.Status == UnitStatus.Borrowed);
    Contract.EnsuresOnThrow<Exception>(u.Status == Contract.OldValue(u.Status));
    ...
  }
  ...
}
```

Ošetření výjimkami (`Requires<T>`, `EndContractBlock`)

Metoda `Contract.EndContractBlock` slouží k označení bloku ověření vstupních podmínek tvaru `if-throw`. To umožňuje ošetřit podmínku vyvoláním specifického typu výjimky, jako například `ArgumentNullException` nebo `ArgumentOutOfRangeException`, a to i v případě, že nejsou použity nástroje Code Contracts. Pokud jsou, lze použít také `Contract.Requires` s generickým parametrem určujícím typ výjimky. Ačkoli vyhodnocení výrazu OCL může způsobit výjimku (reprezentovanou hodnotou `invalid`), nelze v OCL vyhodit výjimku určitého typu, tento způsob tedy dále neuvažujeme.

```
bool Matches(string query){
  if (query == null)
    throw new ArgumentNullException();
  Contract.EndContractBlock();
  ...
}
```

Návratová hodnota (`Result`, `ValueAtReturn`)

V `Contract.Ensures` lze získat návratovou hodnotu příslušné metody pomocí `Contract.Result` a hodnoty výstupních parametrů pomocí `Contract.ValueAtReturn`. V OCL je možný přístup k návratové hodnotě i výstupním parametrům přes proměnnou `result`.

Předchozí hodnoty (`OldValue`)

Metoda `Contract.OldValue` slouží uvnitř výstupní podmínky, která se vyhodnocuje při návratu z metody, k získání hodnoty výrazu z doby před provedením těla metody. Jelikož v době kontroly následných podmínek již není k dispozici stav

objektů z počátku volání, je celý výraz vypočítán před voláním metody a jeho hodnota tedy nemůže záviset na hodnotách, které jsou známé až po provedení těla metody (např. `Contract.Result` nebo parametry lambda výrazů). V jazyce OCL k podobnému účelu slouží modifikátor `@pre`.

Kvantifikátory (`ForAll`, `Exists`)

Statické metody `Contract.ForAll` a `Contract.Exists` slouží pro vyjádření univerzálního a existenčního kvantifikátoru. Jsou ekvivalentní metodám `System.Linq.Enumerable.All`, `System.Linq.Enumerable.Any`, tedy prvním parametrem je kolekce libovolného typu a druhým parameterm delegát[3], který vyhodnocuje podmínku pro každý prvek kolekce. Metody `Forall` a `Exists` mají navíc přetížení, ve kterém první dva parametry určují rozsah celých čísel.

```
⇒ Contract.ForAll(1, 3, i=>i<4) // Iterace přes číselný rozsah
```

V OCL lze kvantifikátory na kolekci hodnot vyjádřit pomocí iterátorů `forAll` a `exists`. Pro iteraci přes číselný rozsah tento rozsah uvedeme do konstruktoru kolekce, přes niž potom iterujeme.

```
⇒ Sequence{1..3}->forAll(i|i<4) -- Konstrukce kolekce z číselného rozsahu
```

Podmínky v kódu (`Assert`, `Assume`)

Metoda `Contract.Assert` umožňuje ověřit platnost podmínky kdekoli v kódu metody. Metoda `Contract.Assume` se za běhu programu chová stejně jako `Contract.Assert`, rozdíl je při statické analýze, kde tuto podmínku není potřeba dokazovat, ale je brána jako fakt. V OCL je sice možné definovat těla metod, ale pouze jako výraz, tyto metody tedy v jazyce OCL nemají obdobu.

Invarianty (`Invariant`)

Invarianty se za běhu kontrolují při návratu z každé veřejné metody objektu. Zapisují se do metod označených atributem `[ContractInvariantMethod]`, které mohou obsahovat pouze posloupnost volání metody `Contract.Invariant`. Pokud dojde v rámci vyhodnocování invariantu k volání veřejné metody stejné třídy, nejsou invarianty ověřovány znovu, nedojde tedy k nekonečné rekurzi. OCL podporuje invarianty konstrukcí `inv`.

5.4 Atributy

Code Contracts definují několik atributů ve jmenném prostoru třídy `System.Diagnostics.Contracts`. Kromě již uvedeného `ContractInvariantMethodAttribute` uvedeme 3 atributy důležité pro specifikaci integirtních omezení.

[3] Vhodné je využití lambda výrazu.

Metody s `PureAttribute`

Z podmínek integriních omezení lze volat pouze metody, které nemění globální stav. Tuto vlastnost je potřeba vyznačit atributem `[Pure]` na metodě nebo třídě, která ji obsahuje. Podobné omezení je i v OCL, všechny volané operace musejí mít příznak `isQuery`.

Kontrakty pro rozhraní a abstraktní metody `ContractClassAttribute`, `ContractClassForAttribute`

Jelikož rozhraní v jazyce C# nemohou obsahovat těla metod, je nutné všechna integritní omezení pro rozhraní zapsat do zvláštní pomocné třídy. Třída musí být abstraktní a implementovat toto rozhraní. Navíc je pak s rozhraním propojena pomocí atributů `[ContractClass]` a `[ContractClassFor]`. Tento postup lze využít i pro specifikaci kontraktů pro abstraktní metody tříd.

```
[ContractClass(typeof(ISearchableContracts))] // Odkaz na kontrakty
interface ISearchable{
  bool Matches(string query);
}
[ContractClassFor(typeof(ISearchable))] // Odkaz zpět na rozhraní
abstract class ISearchableContracts : ISearchable{
  public bool Matches(string query){ // Kontrakty pro metodu rozhraní
    Contract.Requires(query.Length >= 3);
    ...
  }
}
```

6. Porovnání jazyka OCL s jazykem C# a technologií Code Contracts

V této kapitole jsou představeny jazykové konstrukce jazyka OCL a jeho standardní knihovna. K jednotlivým bodům hledáme odpovídající vyjádření v jazyce C# s technologií Code Contracts. V některých případech je korespondence přímočará a konstrukce má v obou jazycích stejnou syntaxi a sémantiku. Často však narazíme na rozdíly - může jít o jiné pojmenování nebo jinou syntaxi. Naopak podobně vypadající konstrukce se někdy liší v chování v okrajových případech, nebo mají úplně jinou sémantiku. Toto je předvedeno na krátkých ukázkách zdrojových kódů jazyka OCL a C#.

6.1 Integritní omezení jazyka OCL

Integritní omezení jazyka OCL pro model UML lze zachytit buď přímo v modelu elementem `Constraint`, nebo zvlášť v souvislém zdrojovém kódu (*skriptu*). Skript obsahuje deklarace kontextu (`context`), které udávají, k jakému typu, vlastnosi nebo operaci se váží následující integritní omezení. Existuje celkem 7 druhů integritních omezení [OCL, kapitola 12], každé uvozené klíčovým slovem, po kterém zpravidla následuje výraz.

6.1.1 Invarianty (`inv`)

Invariant (`inv`) je integritní omezení pro typ. Je to podmínka typu `Boolean`, která musí být splněna pro všechny jeho instance, může být ovšem dočasně porušen během volání operace na dané instanci. Program eXolutio umožňuje jako nestandardní rozšíření navíc připojit chybovou zprávu.

```
context CalendarDate
  inv: month > 0 and month <= 12
  message: 'Invalid month number'
```

V Code Contracts se invarianty zapisují voláními statické metody `Contract.Invariant`, která jsou umístěna do metod označených atributem `[ContractInvariantMethod]`. Zpráva je druhý nepovinný parametr.

```
struct CalendarDate{
  private readonly int month;
  ...
  [ContractInvariantMethod]
  private void Invariants(){
```

```
    Contract.Invariant(month > 0 && month <= 12, "Invalid month number");
    ...
  }
}
```

6.1.2 Vstupní podmínky (`pre`)

Vstupní podmínka (`pre`) je integritní omezení pro operace. Je to podmínka typu `Boolean`, která musí být splněna při každém volání operace. Může se odkazovat na objekt na kterém je operace volána a na parametry operace (kromě výstupních).

```
context User::return(u:Unit)
  pre: u.status = UnitStatus::Borrowed
```

V Code Contracts se vstupní podmínky vyjadřují voláním statické metody `Contract.Requires` na začátku metody.

```
void Return(Unit u, CalendarDate date){
  Contract.Requires(u.Status == UnitStatus.Borrowed);
  ...
}
```

6.1.3 Výstupní podmínky (`post`)

Výstupní podmínka je též integritním omezením pro operace, ovšem musí být splněna při návratu z volání. Výstupní podmínky se oproti podmínkám vstupním mohou odkazovat též na návratovou hodnotu a výstupní parametry operace přes proměnnou `result`. Mohou dokonce pomocí modifikátor `@pre` přistupovat k hodnotám objektů z doby před provedením operace.

```
context User::return(u:Unit)
  post: u.status = UnitStatus::Returned
```

V Code Contracts se výstupní podmínky zapisují voláním statické metody `Contract.Ensures` na začátku metody. Argumentem je podmínka, která může přistupovat k návratové hodnotě pomocí `Contract.Result`, k výstupním parametrům pomocí `Contract.ValueAtReturn` a k hodnotám z doby před provedením metody pomocí `Contract.PreviousValue`.

```
void Return(Unit u, CalendarDate date){
  ...
  Contract.Ensures(u.Status == UnitStatus.Returned);
  ...
}
```

OCL nerozlišuje, zda volání skončilo běžným návratem nebo výjimkou, proto by stejná podmínka měla být ověřena i v `Contract.EnsuresOnThrow<Exception>`, ovšem s tím rozdílem, že hodnota proměnné `result` je zde `invalid`.

6.1.4 Těla operací (`body`)

Jazyk OCL též umožňuje přímo zapsat *tělo operace* jako výraz `body`.

```
context Title::matches(query:String):Boolean
  body: name.indexOf(query)>0
```

V C# metodu implementujeme jedním příkazem `return`. Pokud ovšem metoda má výstupní parametry, pak tělem výrazu je n-tice obsahující jak návratovou hodnotu, tak hodnoty výstupních parametrů.

```
public class Title : ISearchable{
  [Pure]
  public bool Matches(string query){
    return name.IndexOf(query) > -1;
  }
  ...
}
```

6.1.5 Odvozené hodnoty vlastností (`derive`)

Odvozená hodnota (`derive`) je obdobou `body`, ovšem pro vlastnosti.

```
context Loan::duration
  derive: CalendarDate::difference(start, end)
```

V jazyce C# daný výraz zapíšeme do příkazu `return` v getteru a setter vynecháme.

```
class Book {
  public int Id { get{ return CalendarDate.Difference(start, end); } }
}
```

6.1.6 Výchozí hodnoty vlastností (`init`)

Jazyk OCL umožňuje zadat výchozí hodnotu vlastnosti jako výraz.

```
context Title::maxLoanDate : Integer
  init: 30
context Title::finePerDay : Real
  init: 2.5
```

V C# můžeme výchozí hodnotu vlastnosti nastavit přímo v inicializátoru datové položky, nebo v konstruktoru. Tyto dva přístupy se liší pořadím, ve kterém jsou výrazy vyhodnoceny.

```
public class Title{
  private double finePerDay = 2.5; // Inicializace datové položky
  public int FinePerDay{
    get {return finePerDay;} set {finePerDay = value;}
  }
  public int MaxLoanDate{ get; set; }
  public Titlenit(){
    MaxLoanDate = 30; // Inicializace v konstruktoru
  }
}
```

6.1.7 Definice operací a vlastností (`def`)

Pro použití ve výrazech jazyka OCL lze definovat (`def`) nové vlastnosti a operace, které nejsou v modelu. Přídáním klíčového slova `static` definujeme statické vlastnosti a operace.

```
context CalendarDate
  def: dayInYear:Integer =
    day+Set{1..month-1}->collect(m|monthDays(m,year))->sum()
  static def: yearDays(y:Integer):Integer =
    if isLeapYear(y) then 366 else 365 endif
```

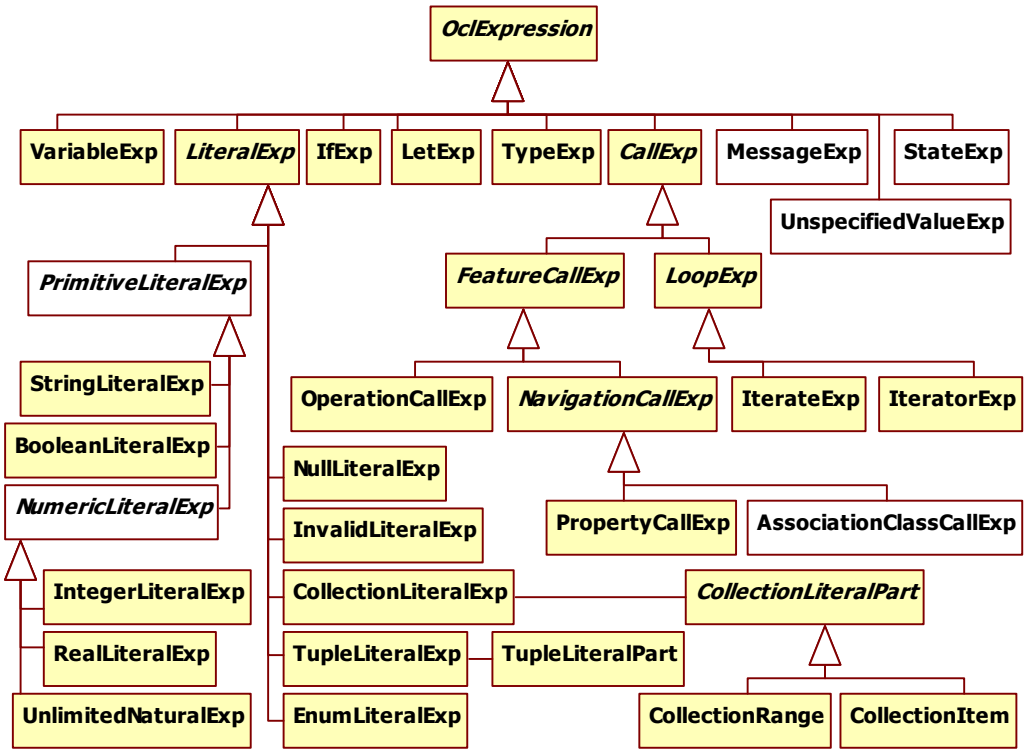
Definované vlastnosti v C# vyjádříme stejně jako vlastnosti s omezením `init`, nebo `body` u operací. Definované metody musíme označit atributem `Pure`.

```
class CalendarDate{
  public int DayInYear{ get {
    CalendarDate self = this;
    return self.day + Enumerable.Range(1, self.month-1).
      Select(m=>MonthDays(m, self.year)).Sum();
  } }
  [Pure]
  public static int YearDays(int y){
    return IsLeapYear(y) ? 366 : 365;
  }
}
```

6.2 Výrazy jazyka OCL

Hlavní částí každého integritního omezení v OCL je výraz. Výraz může podobně jako v programovacích jazycích obsahovat operátory, volání operací, čtení vlastností objektů, přístupy k lokálním proměnným a další konstrukce. Všechny druhy výrazů jsou uvedeny na obrázku 6.1 a popsány v kapitole Abstract Syntax specifikace [OCL, kapitola 8.3]. Výrazy označené bíle nejsou v programu eXolutio a této práci podporovány[1].

[1] V programu eXolutio je navíc jako nestandardní rozšíření výraz `ClassLiteralExp`, v této práci není uvažován.



Obrázek 6.1: Abstraktní syntaxe jazyka OCL. Zdroj: [OCL, kapitola 8]

Každý výraz má *známý typ* (nebo též *statický typ*), který je určen při parsování podle pravidel pro jednotlivé druhy výrazů. Při vyhodnocení má potom výraz *hodnotu*, jejímuž typu říkáme *skutečný typ*. Skutečný typ pak musí být *kompatibliní* se známým typem. Kompatibilita typů je popsána v [OCL, kapitola 8.2.1].

6.2.1 Operátory `::`, `.` a `->`

Operátory `::`, `.` a `->` najdeme v mnoha programovacích jazycích, kde představují různé způsoby přístupu ke členům jmenných prostorů, tříd a podobných konceptů. Výjimkou nejsou ani jazyky OCL a C#.

V jazyce OCL používáme operátor `.` k volání operací (`OperationCallExp`) a přístupu k vlastnostem objektu (`PropertyCallExp`). Pokud je však statický typ operandu kolekce, neprovede se volání (nebo přístup k vlastnosti) na kolekci samotné, ale jednotlivě na prvcích kolekce; zápis je totiž ekvivalentní přístupu k vlastnosti proměnné v iterátoru `collect`[2].

Operátor `->` používáme k volání operací (`PropertyCallExp`) a iterátorů (`LoopExp`) pracujících s kolekcemi. Pokud výraz na levé straně operátoru nemá typ kolekce,

[2]Např. `Set{-1,0,1}.abs()` je ekvivalentní `Set{-1,0,1}.collect(i|i.abs())`.

ale jiný typ `T`, použije se na něj implicitně operace `oclAsSet` a přístup se provede na výsledné kolekci typu `Set(T)`.

Operátor `::` používáme v literálech výčtových typů (`EnumLiteralExp`), přístupech ke statickým vlastnostem nebo operacím, při přístupu k předefinovaným vlastnostem předka a v názvech balíků.

```
⇒ user.name -- Čtení vlastnosti objektu
⇒ user.Loan.Unit -- Implicitní iterátor collect
⇒ UnitStatus::Borrowed -- Výčtová konstanta
⇒ cd.Title::matches(query) -- Předefinovaná operace
⇒ user.Loan->size() -- Operace na kolekci
⇒ loan.end->isEmpty() -- Implicitní operace oclAsSet
```

V C# používáme k přístupu ke všem metodám, vlastnostem a datovým položkám operátor `.`. Operátory `->` a `::` se používají pouze zřídka a to pro přístup k datovým položkám struktury přes ukazatel a do jmenného prostoru přes alias.

```
⇒ book.Name // Čtení vlastnosti
⇒ UnitStatus.Borrowed // Výčtová konstanta
⇒ base.Matches(query) // Přístup ke členu rodičovské třídy
using Alias = System.Collections.Generic; // Deklarace aliasu
Alias::List<int> i; // Přístup do jmenného prostoru
unsafe{
 CalendarDate c = new CalendarDate( ... );
 CalendarDate* p = &c; // Deklarace ukazatele
⇒  p->year // Čtení vlastnosti přes ukazatel
}
```

6.2.2 Volání operací

V OCL při volání operací, ať už z UML modelu nebo standardní knihovny, argumenty píšeme do závorek za název operace. V jazyce UML ovšem mohou mít operace i výstupní parametry. OCL pro ně nemá žádou zvláštní syntaxi. Při volání výstupní parametry vynecháme a návratová hodnota funkce ve výrazu je potom n-tice (`Tuple`), která obsahuje všechny výstupní a vstupně-výstupní parametry a návratovou hodnotou funkce (pokud není typu `OclVoid`) v položce `result`.

```
Library::availableUnit(t:Title, date:CalendarDate, status:UnitStatus) : Unit
  ...
⇒ lib.availableUnit(title, date).result -- Volání operace s výstupním parametrem
⇒ lib.availableUnit(title, date).status -- Čtení výstupního parametru
```

V C# jsou výstupní a vstupně-výstupní parametry uloženy do proměnných a je potřeba zadat klíčové slovo `ref` nebo `out` jak v deklaraci metody, tak při jejím volání.

```
Unit AvailableUnit(Title t, ref CalendarDate date, out UnitStatus status);
  ...
⇒ library.AvailableUnit(title, ref date, out status);
```

6.2.3 Předchozí hodnoty (`@pre`)

Pokud potřebujeme vyjádřit vztah návratové hodnoty nebo stavu po provedení operace se stavem před provedením operace, musíme jej zapsat do výstupní podmínky, která je vyhodnocena až po provedení operace. Přístup k předchozí hodnotě vlastnosti nebo operace objektu pak označíme modifikátorem `@pre`. Je potřeba dát pozor na to, že se modifikátor nevztahje na celý podvýraz vlevo, ale pouze na jeden přístup, tedy identita objektu, ke kterému se přistupuje, je normálně vyhodnocena ve stavu po provedení operace.

```
⇒ book@pre.name@pre -- Předchozí hodnota vlastnosti objektu,
    -- jehož instance je vypočtena před voláním
⇒ book@pre.name -- Současná hodnota vlastnosti objektu,
    -- jehož instance je vypočtena před voláním
⇒ book.name@pre -- Předchozí hodnota vlastnosti objektu,
    -- jehož instance je vypočtena po volání
```

Code Contracts umožňuje před provedením metody vyhodnotit pouze celý výraz. Zapíšeme ho jako argument metody `Contract.PreviousValue` (již lze použít pouze v argumentu metody `Contract.Ensures`). V Code Contracts tedy lze reprezentovat pouze ty výrazy z OCL, kde se v žádném povýrazu přístupu k vlastnosti nebo volání operace s modifikátorem `@pre` nenachází přístup bez `@pre`.

```
⇒ Contract.PreviousValue(this.book.name) // Předchozí hodnota vlastnosti objektu
⇒ Contract.PreviousValue(this.book).name // Současná hodnota vlastnosti objektu
```

6.2.4 Lokální proměnné (`let`)

Pro lepší strukturování kódu nebo v případě, že potřebujeme hodnotu výrazu použít vícekrát, se hodí konstrukce `let` (`LetExp`), která umožňuje vytvořit lokální proměnnou s daným typem a hodnotou.

```
⇒ let x : Integer = 3 in x * 10 -- 30
```

Jazyk C# neumožňuje deklarovat nové proměnné uvnitř výrazu. Toto omezení lze ale obejít použitím lambda výrazu: proměnná je jeho parametr, hodnotu proměnné předáme jako argument. Pokud navíc chceme, aby případné výjimky při vyhodnocení výrazu přiřazeného do proměnné byly vyhozeny až na místě použití proměnné (tak se chová hodnota `invalid` v OCL), místo hodnoty samotné předáme další lambda výraz bez parametrů a vyhodnotíme ho až na místě použití.

```
TResult Let<TVar,TResult>(TVar variableValue, Func<TVar,TResult> inExpression){
        return inExpression(variableValue);
}
⇒ Let<int,int>(3, x => x * 10) // 30
⇒ Let<Func<int>,int>(()=>3, x => x() * 10) // 30
```

6.2.5 Podmínky (`if-then-else-endif`)

Výraz `IfExp` předsatvuje jednoduché větvení podle podmínky typu `Boolean`. Pokud je vyhodncena na `true`, vyhodnotí se výraz za `then`, pokud je `false`, vyhodnotí se výraz za `else`; jinak je výsledek `invalid`.

```
⇒ if true then 'ok' else 'not ok' endif -- 'ok'
```

V C# se úplně stejně chová podmínkový operátor `?:`. Pokud jsou ovšem typy obou větví různé, musí mezi nimi existovat implicitní konverze.

```
⇒ true ? "ok" : "not ok" // "ok"
⇒ true ? "ok" : 1 // chyba překladu
```

6.2.6 N-tice (`Tuple`)

Pokud v OCL potřebujeme sdružit více hodnot do jedné, můžeme použít typy `Tuple`. Typ takové n-tice je jednoznačně určen názvy a typy jejích částí; na pořadí nezáleží. Instanci vytvoříme pomocí literálu `TupleLiteralExp`.

```
⇒ Tuple{i:Integer = 1, s:String = 'str'}
```

Typy `Tuple` jsou instancemi metatypu `TupleType`, jenž je potomkem `DataType`, tedy dvě n-tice jsou si rovné, pokud jsou stejného typu a obsahují stejné hodnoty.

Knihovna BCL obsahuje od verze .NET Framework 4 sadu generických tříd `System.Tuple`. Tyto třídy ovšem představují n-tice v jiném smyslu, jelikož jednotlivé složky nejsou pojmenované a rozlišují se jen pořadím.

Podobnou syntaxi jako literály `Tuple` mají v C# anonymní typy. Překladač vygeneruje novou nepojmenovanou třídu se specifikovanými vlastnostmi. V rámci jedné assembly jsou anonymní typy se stejnými názvy, typy i pořadím vlastností stejné. Jelikož je typ anonymní, nelze jej v programu nijak explicitně vyjádřit, musíme spoléhat na inferenci typů. Anonymní typy podporují porovnávání po složkách (metody `Equals` a `GetHashCode`). Jelikož samotný kód v jazyce C# také generujeme, mohli bychom pro n-tice generovat neanonymní třídy, přibližně ekvivalentní kódu, který by překladač vygeneroval pro anonymní typ.

Korektní chování, kde nezáleží na pořadí prvků, získáme jejich uložením do slovníku typu `Dictionary`.

```
Tuple<int, string> tuple = new Tuple<int, string>(1, "str"); // Typ Tuple
var tuple = new {i = 1, s = "str"}; // Anonymní typ
public class Tuple_i_Integer_s_String{ // Vygenerovaný typ
  public Tuple_i_Integer_s_String(int i, string s){
    this.i = i; this.s = s;
  }
  public int i; public string s;
}
var tuple = new Dictionary<string, object>{{"i",1},{"s","str"}}; // Slovník
```

6.2.7 Zasílání zpráv (^, ^^)

Výrazy `MessageExp` zjišťují, zda byla v průběhu operace zavolána jiná operace na daném objektu. Operátor `^` vrací příznak typu `Boolean`, operátor `^^` vrací kolekci všech volání, ze které pak lze získat jejich argumenty. Do jisté míry tím možňuje obejít omezení, že z výrazů jazyka OCL lze volat pouze operace s atributem `isQuery`. Implementace by pravděpodobně vyžadovala zaznamenávání všech volání v operaci a v Code Contracts podobný koncept neexistuje, tyto operátory tedy v této práci implementovány nebudou.

6.2.8 Přístup k elementům modelu

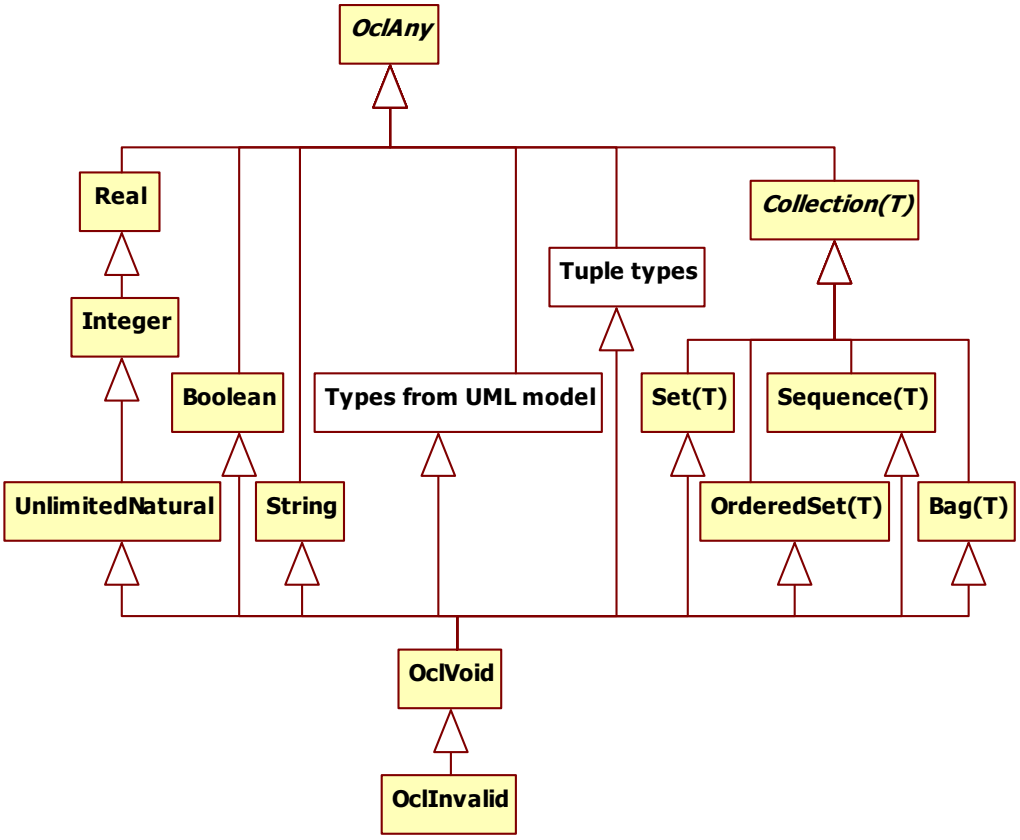
Jako hodnota výrazu v OCL se může vyskytovat také typ. Typem takové hodnoty je pak `Classifer` z metamodelu UML. Buď můžeme typ zapsat přímo (`TypeExp`), nebo můžeme získat skutečný typ hodnoty operací `oclType` ze standardní knihovny. Naopak operace `oclAsType`, `oclIsTypeOf` a `oclIsKindOf` mají typ jako parametr. Typy, které mají omezený počet instancí, mají operaci `allInstances`, která vrací jejich množinu.

```
⇒ 'str'.oclType() -- String
⇒ 'str'.oclAsType(OclAny) -- 'str'
⇒ 1.oclIsKindOf(Real) -- true
⇒ Boolean.allInstances() -- Set{true, false}
```

6.3 Standardní knihovna jazyka OCL

Kromě typů definovaných uživatelem v UML modelu lze ve výrazech jazyka OCL pracovat s typy ze standardní knihovny, která obsahuje:

- primitivní typy (`Boolean`, `String`, `Integer`, `Real`, `UnlimitedNatural`), jež odpovídají primitivním typům z UML,
- kolekce (`Collection`, `Sequence`, `Set`, `Bag`, `OrderedSet`),
- další typy (`OclAny`, `OclVoid`, `OclInvalid`, `OclMessage`).



Obrázek 6.2: Typy jazyka OCL a jejich kompatibilita

Standardní knihovna jazyka OCL je popsána v kapitole 11 specifikace [OCL]. Sémantika typů a jejich operací není v této kapitole popsána zcela formálně, pouze u některých operací je výsledek specifikován pomocí výstupních podmínek.

V následujích odstavcích jsou představeny jednotlivé typy ze standardní knihovny jazyka OCL a popsány jejich operace. U každého typu stadnardní knihovny najdeme vyjádření odpovídajícího konceptu v jazyce C# a knihovně BCL a popíšeme případné rozdíly mezi nimi.

6.3.1 Univerzální typ (`OclAny`)

Typ `OclAny` stojí v OCL na vrcholu typové hierarchie, tedy všechny typy jsou s ním kompatibilní[3] a do proměnné typu `OclAny` lze přiřadit libovolnou hodnotu.

```
⇒ let a:OclAny = 4 in a -- Přiřazení do proměnné OclAny
```

Stejný koncept univerzálního typu je i jazyce C#, kde všechny typy dědí od typu `object` (`System.Object`). Do proměnné typu `object` lze tedy přiřadit hodnotu

[3] Ve starších verzích nebyl `OclAny` předek kolekcí ([RTF, Issue 12948] http://www.omg.org/issues/ocl2-rtf.html#Issue12948)

jakéhokoli typu. U hodnotových typů dojde k boxingu, tedy je naalokován nový objekt a hodnota je zkopírována.

```
object x = 1;
```

6.3.2 Neplatné hodnoty (`OclInvalid`)

Hodnota `invalid` je vrácena některými operacemi v situacích jako dělení nulou, volání operace na hodnotě `null`, nebo snaha přetypovat hodnotu na nekompatibilní typ. Hodnota `invalid` je jediná instance typu `OclInvalid`, jenž je v jazyce OCL kompatibilní se všemi typy, tedy výraz libovolného typu může nabývat hodnoty `invalid`. Pokud se pokusíme přistupovat k atributu nebo volat operace na `invalid`, případně volat operace s argumentem `invalid`, výsledkem je zase `invalid`. Výjimku tvoří operace `oclIsUndefined`, `oclIsInvalid` a booleovské operace `and`, `or` a `implies`. Hodnota `invalid` může být také uvedena ve výrazu jako literál.

```
⇒ let i:Integer = 0 in 1.div(i) -- invalid (dělení nulou)
⇒ let j:Integer = null in j + 2 -- invalid (volání operace na hodnotě null)
⇒ let o:OclAny = 1 in o.oclAsType(String) -- invalid (přetypování)
⇒ invalid -- invalid (literál)
```

V jayzce C# v těchto případech dojde k vyvolání výjimky. Výjimky jsou objekty typu odvozeného od `System.Exception`, skutečný typ výjimky se liší podle důvdou výjimky.

```
int i = 0;
⇒ 1/i // DivideByZeroException (dělení nulou)
Book b = null;
⇒ b.pages // NullReferenceException (hodnota null)
object o = 3;
⇒ (string) o // InvalidCastException (přetypování)
throw new Exception(); // Exception (příkaz throw)
```

Při vyhození výjimky dojde k zastavení vyhodnocování výrazu a řízení je předáno bloku `catch`, který zpracovává daný typ výjimky. Výjimka se tedy šíří podobně jako hodnota `invalid` v OCL. Pokud ovšem chceme zastavit šíření výjimky (např. pro implementaci operace `oclIsInvalid`), nemůžeme použít `catch` blok uvnitř výrazu. Můžeme však výraz, jehož výjimky chceme zachytit, umístit do lambda funkce a jako delegáta jej předat metodě, která jej vyhodnotí v `catch` bloku.

```
⇒ let x=1 in (1/x).oclIsInvalid() -- false
⇒ let y=0 in (1/y).oclIsInvalid() -- true
```

```
bool IsInvalid(Func<object> expr){
        try{
                expr(); // Vyhodnotit výraz
                return false; // Výraz byl vyhodnocen
        }
        catch(Exception){
                return true; // Došlo k výjimce
        }
```

```
}
int x = 1;
⇒ IsInvalid(()=>(1/x)) // false
int y = 0;
⇒ IsInvalid(()=>(1/y)) // true
```

6.3.3 Typ prázdné hodnoty (`OclVoid`)

Typ `OclVoid` je v jazyce OCL kompatibilní se všemi typy kromě typu `OclInvalid`. Jeho jedinou instancí je hodnota `null`. Podobně jako u `invalid`, volání bežných operací na hodnotě `null` vrací hodnotu `invalid`. Pro ošetření hodnoty `null` lze použít operaci `oclIsUndefined` nebo `isEmpty` na kolekci vrácené implicitním voláním operace `oclAsSet`.

```
⇒ null -- null (literál)
⇒ let v:OclVoid = null in v.oclIsUndefined() -- true
⇒ let v:OclVoid = null in v->isEmpty() -- true (v.oclAsSet()->isEmpty())
```

V C# také existuje literál `null`. Literál `null` lze přiřadit do proměnné libovolného referenčního typu, ovšem sám o sobě nemá žádný typ. V C# je při zavolání metody na hodnotě `null` vyvolána výjimka `NullReferenceException`. Mnoho metod vyžaduje, aby jejich parametry nenabývaly hodnot `null`, v takovém případě je běžné tuto situaci ošetřit vyhozením výjimky `ArgumentNullException`.

```
string n = null; // Literál null
⇒ n.EndsWith("e"); // NullReferenceException
⇒ "e".EndsWith(n); // ArgumentNullException
```

U metod, které nevracejí žádnou hodnotu, v C# specifikujeme `void` jako návratový typ metody. Nelze ovšem například vytvořit proměnnou typu `void` ani instanci typu `void`. Narozdíl od OCL není mezi hodnotou `null` a typem `void` žádný vztah.

```
void Method(){...} // Metoda bez návratové hodnoty
```

6.3.4 Zprávy (`OclMessage`)

Typ `OclMessage` se týká zasílání zpráv a není v této práci uvažován.

6.3.5 Operace definované na typech `OclAny`, `OclInvalid` a `OclVoid`

Operace definované na typu `OclAny` lze použít na libovolné hodnotě. Tyto operace často vyžadují zvláštní zpracování hodnot `null` a `invalid`, proto jsou na typech `OclVoid` a `OclInvalid` předefinovány.

Porovnání na rovnost (=, <>)

Typ `OclAny` definuje operace `=(o:OclAny):Boolean`, a `<>(o:OclAny):Boolean`, což umožňuje každé dvě hodnoty porovnávat na rovnost. Instance objektu je rovna sama sobě, u typů, které jsou instancemi `DataType` jsou si navíc rovny objekty se stejným typem a stejnou hodnotou.

V C# máme více možností, jak porovnávat hodnoty. Statická metoda `object.ReferenceEquals` porovnává reference, tedy dvě instance si nejsou nikdy rovny. Toto porovnání není užitečné pro hodnotové typy, neboť při konverzi hodnotového typu na object dochází k boxingu, tedy vzniká nová instance.

Typ `object` má virtuální metodu `Equals(object)`, která se pro referenční typy chová stejně jako `ReferenceEquals`, pro hodnotové typy však porovnává typ a obsah datových položek. Důležité je, že třídy a struktury mohou tuto metodu předefinovat (například pokud chceme aby u referenčního typu byly porovnávány jeho datové položky). Spolu s metodou `Equals` je potřeba předefinovat i metodu `GetHashCode`, která musí pro objekty, které si mají být rovny, vrátit stejné celé číslo.

Pokud by hodnota, kterou porovnáváme, byla `null`, volání instanční metody by vyvolalo výjimku `NullReferenceException`. Proto je vhodnější použít staickou metodu `object.Equals`, která nejprve porovná reference a pouze pokud jsou různé a ani jedna z nich není `null`, zavolá instanční metodu `Equals`.

Typy mohou také implementovat rozhraní `IEquatable<T>` a jeho metodu `Equals(T)`, kde `T` je ten samý typ, ve kterém je metoda implementována. Používání této metody nám umožňuje vyhnout se konverzi (a případnému boxingu) na `object`, jinak by měla být ekvivalentní `Equals(object)`.

Operátor `==` definovaný pro typ `object` porovnává reference (jako `ReferenceEquals`), ovšem jiné typy ho mohou přetěžovat (např `string`, `int`). Typ, který přetěžuje `==`, by měl také předefinovat metody `Equals` a `GetHashCode`, opačně to vyžadováno není. Zároveň s přetížením operátoru `==` je vyžadováno také přetížení operátoru `!=`, který odpovídá operátoru `<>` v OCL.

```
class Point{
        public int X{get;set;}
        public int Y{get;set;}
        public override int GetHashCode(){
                return X ^ Y;
        }
        public override bool Equals(object o){
                Point pt = o as Point;
                if((object)pt == null)
                        return false;
                return X == o.X && Y == o.Y:
        }
}
Point a = new Point();
Point b = new Point();
⇒object.ReferenceEquals() // false (porovnání referencí)
```

```
⇒object.Equals(a, b) // true (metoda Equals)
⇒(object)a == (object)b // false (porovnání referencí)
⇒a == b // true (operátor ==)
```

Ošetření hodnot `null` a `invalid`

Operace `oclIsInvalid():Boolean`, a `oclIsUndefined():Boolean` jsou jedny z mála operací, která i při volání na hodnotě `invalid` nevracejí `invalid`. Umožňují tedy zastavit propagaci hodnoty `invalid`, podobně jako v jazyce C# konstrukce `catch`.

x	x.oclIsInvalid()	x.oclIsUndefined()	x=null	x->isEmpty()
null	false	true	true	true
invalid	true	true	invalid	invalid
jiná hodnota	false	false	false	false

Tabulka 6.1: Testování hodnot `invalid` a `null` v OCL

Jazyk C# neumožňuje zachytit výjimku uvnitř výrazu, díky lambda výrazům je ale možné výraz vyhodnotit v `catch` bloku uvnitř metod implementujících `oclIsInvalid` a `oclIsUndefined`.

```
int i = 0;
⇒Ocl.IsInvalid(()=>1/i) // true (vyhodnoceno v catch bloku uvnitř metody)
```

K porovnání s hodnotou `null` použijeme `ReferenceEquals` nebo operátor `==`.

Přetypování (`oclAsType`)

Jazyky OCL a C# jsou staticky typované. Pomocí operace `oclAsType (type:Classifier):T` můžeme v OCL změnit statický typ výrazu. Pokud je při vyhodnocení výrazu hodnota kompatibilní s požadovaným typem, vrací `oclAsType` stejnou hodnotu, v případě nekompatibility vrací `invalid`. Tato operace je zvláštní tím, že její návartový typ (`T`) závisí na hodnotě jejího parametru. Parser jazyka OCL v programu eXolutio proto vyžaduje, aby argumentem byla typová konstanta (výraz `TypeExp`). Popora obecného výrazu by výrazně zesložitila konstrukci parseru.

```
⇒let o:OclAny=1 in o.oclAsType(Integer) + 1 -- 2
⇒let o:OclAny='2' in o.oclAsType(Integer) + 1 -- invalid (nekompatibilní typ)
```

V jazyce C# plní tuto úlohu přetypování. Narozdíl od OCL není možné přetypovávat libovolné typy. U referenčních typů lze přetypovávat na předky a potomky. Pokud typ hodnoty není s cílovým typem kompatibilní, způsobí přetypování výjimku `InvalidCastException`. Přetypování se také používá ke konverzím (u primitivních typů nebo uživatelsky definované konverze). Operátor `as` přetypovává pouze referenční nebo nullable[4] typy a v případě neúspěchu vrací `null`.

[4]Nullable typy, `System.Nullable<T>` nebo `T?`, jsou hodnotové typy, umožňující reprezentovat jak hodnoty typu `T`, tak hodnotu `null`.

```
object o = 1, p = "1";
⇒ (int)o + 1; // 2
⇒ (int)p + 1; // InvalidCastException (nekomatibilní typ)
⇒ (o as int?) + 1; // 2
⇒ (p as int?) + 1; // null (nekomatibilní typ)
```

Zjištění typu hodnoty (`oclType`)

Zjistit skutečný typ hodnoty můžeme pomocí operace `oclType():Classifier`. Tato operace funguje i pro hodnoty `null` a `invalid` (vrací `OclVoid` resp. `OclInvalid`).

```
⇒ let o:OclAny = 1 in o.oclType() -- Integer
⇒ let p:OclAny = null in p.oclType() -- OclVoid
```

V jazyce C# definuje typ `object` metodu `GetType`, která vrací instanci třídy `System.Type`, reprezetující typ objektu. `GetType` je instanční metoda, takže volání na hodnotě `null` vyvolá výjimku.

```
object o = 1;
⇒ o.GetType(); // System.Int32
object p = null;
⇒ p.GetType(); // NullReferenceException
```

Pokud chceme zjistit, zda má hodnota daný typ, použijeme `oclIsTypeOf(Classifier):Boolean`. Tato operace pro hodnoty `null` a `invalid` vrací vždy `invalid`, čímž se mírně liší od porovnání s hodnotou vrácenou `oclType()`.

```
⇒ let o:OclAny = 1 in o.oclIsTypeOf(Real) -- false (Integer <> Real)
⇒ let p:OclAny = null in p.oclIsTypeOf(OclAny) -- invalid
```

V C# porovnáme typ vrácený metodou `GetType` s operátorem `typeof`.

```
object o = 1;
⇒ o.GetType() == typeof(double); // false
object p = null;
⇒ p.GetType() == typeof(double); // NullReferenceException
```

Operace `oclIsKindOf(Classifier):Boolean` testuje kompatibilitu hodnoty s daným typem. Pro `null` a `invalid` vrací `invalid`.

```
⇒ let o:OclAny = 1 in o.oclIsTypeOf(Real) -- true (je kompatibilní s Real)
⇒ let p:OclAny = null in p.oclIsTypeOf(OclAny) -- invalid
```

oclIsKindOf	OclInvalid	OclVoid	OclAny	Integer
invalid	invalid	invalid	invalid	invalid
null	invalid	invalid	invalid	invalid
1	false	false	true	true
1.3	false	false	true	false

Tabulka 6.2: Operace `isKindOf` v OCL

V jazyce C# se kompatibilita hodoty s typem zjišťuje operátorem `is`. Na objektech `System.Type` můžeme použít metodu `IsAssignableFrom`.

```
object o = 1
⇒ o is int // true
⇒ typeof(int).IsAssignableFrom(o.GetType()) // true
```

x	x is int	x is object	x is double
výjimka	výjimka	výjimka	výjimka
null	false	false	false
1	true	true	false
1.3	false	true	true

Tabulka 6.3: Operátor `is` v jazyce C#

Implicitní konverze na kolekci (`oclAsSet`)

Pokud na levé straně operátoru `->` stojí výraz, jehož statický typ není kolekce, použije se na něj implicitně operace `oclAsSet`, která hodnotu převede na kolekci `Set`, na které je potom provedena operace uvedená na pravé straně operátoru. Pro hodnotu `null` vrátí `oclAsSet` prázdnou množinu, pro ostatní hodnoty jednoprvkovou množinu obsahující tuto hodnotu. (Pro `invalid` je výsledkem zase `invalid`.) To umožňuje jednoduchý test proměnné na `null` v OCL zapsat jako `x->isEmpty()`. Tato operace byla do specifikace doplněna v OCL verze 2.4.

```
⇒ 1->including(2) -- Set{1,2} (1.oclAsSet()->including(2))
```

Objekt vytvořený v průběhu operace (`oclIsNew`)

Operaci `oclIsNew():Boolean` lze použít pouze v po-podmínce operace a zjišťuje, zda byl objekt vytvořen během volání této operace.

```
context Book::createInstance()
post: result.oclIsNew()
```

Jazyk C# ani CodeContracts neposkytují žádný způsob, jak zjistit, kdy byl objekt vytvořen, proto tato operace v této práci není implementována.

Stav objektu (`oclIsInState`)

Operace `oclInState(OclState):Boolean` není ve specifikaci jasně definována[5] a v jazyce C# neexistuje podobná konstrukce, proto tuto operaci v této práci neuvažujeme.

6.3.6 Řetězce (`String`)

Pro práci s řetězci slouží v jazycích UML a OCL primitivní typ `String`. Jeho obdobou je v jazyce C# primitivní referenční typ `System.String`. Zatímco v OCL je řetězec posloupnost znaků bez určeného kódování, v jazyce C# je řetězec je vždy

[5] http://www.omg.org/issues/ocl2-rtf.html#Issue15357

posloupnost 2-bytových znaků `char` (`System.Char`) v kódování UTF-16. V tomto kódování ovšem může 1 znak ze znakové sady Unicode zabírat 2 nebo 4 bajty (znaky s kódem od U+010000 jsou reprezentovány jako takzvaný surrogate pair), takže délka řetězce ve znacích Unicode se tedy může lišit od délky řetězce ve znacích `char`. Metody pro práci s řetězci v BCL vždy pracují s jednotlivými znaky `char`, správnému zpracování surrogate pairs je tedy potřeba věnovat zvláštní pozornost.

V OCL zapisujeme řetězcové literály do jednoduchých uvozovek, v C# do dvojitých.

```
⇒ 'some string' -- Literál typu String
```

```
⇒ "some string" // Literál typu string (System.String)
```

V OCL chování některých operací na typu `String` (například `toUpperCase`, `toLowerCase`, `<`, `<=`, `>`, `>=`) závisí na nastavení národního prostředí. Národní prostředí pro takové operace je určeno hodnotou proměnné `oclLocale` v místě volání metody. Ta může obsahovat až tři části oddělené podtržítky (`'_'`):

- jazyk (dvojmístný kód podle standardu ISO 639),
- země (dvojmístný kód podle standardu ISO 3166),
- varianta (závislé na implementaci).

Výchozí hodnota je `'en_us'`.

V knihovnách .NET mají metody, jejichž chování závisí na národním nastavení, parametr typu `System.Globalization.CultureInfo`. Instance této třídy se získávají voláním statické metody `CultureInfo.GetCultureInfo`, s parametrem udávajícím název národního prostředí, který sestává z kódu jazyka (ISO 639) a země (ISO 3166), oddělených znakem `'-'`, případně může obsahovat navíc kód písma (podle standardu ISO 15924). Na velikosti písmen v kódech nezáleží.

Spojování řetězců (`+`, `concat`)

Operace `+(s:String):String` i `concat(s:String):String` spojují dva řetězce do jednoho.

```
⇒ 'A' + 'B' -- 'AB'
⇒ 'A'.concat('B') -- 'AB'
⇒ 'A'.concat(null) -- invalid
```

Stejně tak v C# můžeme použít jak operátor `+`, tak statickou metodu `Concat`. Pokud je některý z argumentů `null`, je interpretován jako prázdný řetězec.

```
⇒ "A" + "B" // "AB"
⇒ string.Concat("A", "B"); //"AB"
⇒ string.Concat("A", null); //"A"
```

Délka řetezce (`size`)

Operace `size():Integer` vrací délku řetězce ve znacích.

```
⇒ 'string'.size() -- 6
```

Typ `string` má v C# vlastnost `Length`, která vrací délku ve znacích `char` v kódování UTF-16, tedy každý surrogate pair je počítán jako dva znaky. Metoda `LengthInTextElements` typu `System.Globalization.StringInfo` sice počítá surrogate pairs jako jeden znak, stejnou věc ale provede i se znaky složenými z kombinujících diakritických znamének.

```
⇒ "string".Length // 6
⇒ "\uD800\uDC00".Length // 2 (surrogate pair)
⇒ new StringInfo("\uD800\uDC00").LengthInTextElements // 1 (surrogate pair)
⇒ "a\u0301".Length // 2 (kombinující znak)
⇒ new StringInfo("a\u0301").LengthInTextElements // 1 (kombinující znak)
```

Podřetězce (`at`, `substring`)

Operace `at(Integer):String`, vybere z řetězce znak na zadané pozici. Protože OCL nemá zvláštní typ na reprezentaci jednotlivých znaků, je znak vrácen jako řetězec délky 1. Delší podřetězec získáme operací `substring(lower:Integer, upper:Integer):String`. Obě hraniční pozice jsou zahrnuty do výsledku.

```
⇒ 'string'.at(2) -- 't'
⇒ 'string'.substring(2,4) -- 'tri'
```

V jazyce C# lze přistupovat k jednotlivým znakům (typu `char`) řetězce operátorem `[]`. Na rozdíl od OCL jsou znaky číslovány od `0`. Podřetězec získáme instanční metodou `Substring`. Jejím druhým parametrem však není pozice posledního vybraného znaku, ale délka vybraného podřetězce. Opět je potřeba věnovat pozornost surrogate pairs.

```
⇒ "string"[2] // 'r'
⇒ "string".Substring(2,4) // "ring"
```

Konverze velikosti písmen (`toUpperCase`, `toLowerCase`)

Operace `toUpperCase():String`, resp. `toLowerCase():String` vrátí řetězec převedený na velká resp. malá písmena. Způsob převodu závisí na hodnotě `oclLocale`.

```
⇒ 'String'.toUpperCase() -- 'STRING' pro oclLocale='en_us'
⇒ 'String'.toLowerCase() -- 'string' pro oclLocale='en_us'
```

Typ `string` k tomuto účelu obsahuje metody `ToUpper(CultureInfo)` a `ToLower(CultureInfo)`.

```
⇒ "String".ToUpper(CultureInfo.GetCultureInfo("en-us")) // "STRING"
⇒ "String".ToLower(CultureInfo.GetCultureInfo("en-us")) // "string"
```

Vyhlednávání podřetězce (`indexOf`)

Operace `indexOf(s:String):Integer` slouží k vyhledávání podřetězce v řetězci. V případě neúspěchu vrací nulu.

```
⇒ 'string'.indexOf('tri') -- 2
⇒ 'string'.indexOf('a') -- 0
```

Typ `string` obsahuje metodu `IndexOf(String, StringComparison)`, která vrací pozici podřetězce (počítanou od `0`), nebo `-1`, pokud podřetězec neexistuje. Jako druhý parametr uvedeme `StringComparison.Ordinal`, bez uvedení tohoto parametru by vyhledávání záviselo na nastavení národního prostředí.

```
⇒ "string".IndexOf("tri", StringComparison.Ordinal) // 1
⇒ "string".IndexOf("a", StringComparison.Ordinal) // -1
```

Porovnávání řetězců (`<`, `>`, `<=`, `>=`)

Porovnávání řetězců v OCL operátory `<(s:String):Boolean`, `>(s:String):Boolean`, `<=(s:String):Boolean` a `>=(s: String):Boolean` závisí na hodnotě `oclLocale`.

```
⇒ 'a' < 'b' -- true pro oclLocale='en_us'
⇒ 'a' > 'b' -- false pro oclLocale='en_us'
```

V jazyce C# nelze řetězce porovnávat přímo pomocí operátorů `<`, `>`, `<=` a `>=`, je nutné použít statickou metodu `string.Compare`. Třetí argument určuje, že se má brát v úvahu velikost písmen. Návratovou hodnotu potom porovnáme s nulou příslušným operátorem.

```
⇒ string.Compare("a", "b", false, CultureInfo.GetCultureInfo("en-us"))<0 // true
⇒ string.Compare("a", "b", false, CultureInfo.GetCultureInfo("en-us"))>0 // false
```

Porovnání bez rozlišení velikosti písmen (`equalsIgnoreCase`)

Porovnávání řetězců bez ohledu na velikost písmen `equalsIgnoreCase(String):Boolean` závisí na hodnotě `oclLocale`.

```
⇒ 'string'.equalsIgnoreCase('String') -- true pro oclLocale='en_us'
```

Typ `string` umožňuje porovanávat řetězce metodou `Equals`. Tato metoda umožňuje zvolit různé typy porovnání definované ve výčtovém typu `StringComparison`, neumožňuje však přímo zadat jiné než výchozí národní prostředí. Řešením je použít opět medodu `Compare`, tentokrát s `true` na místě třetího argumentu.

```
⇒ string.Equals("string", "String", StringComparison.CurrentCultureIgnoreCase)
    // true
⇒ string.Compare("string", "String", true, CultureInfo.GetCultureInfo("en-us"))
    == 0 // true
```

Konverze na kolekci (`characters`)

Operace `characters():Sequence(String)` rozdělí řetězec na jednotlivé znaky a vrátí posloupnost řetězců obsahujících jednotlivé znaky.

```
⇒ 'string'.characters() -- Sequence{'s','t','r','i','n','g'}
```

Typ `System.String` implementuje rozhraní `IEnumerable<char>`, lze jej tedy přímo použít jako kolekci znaků.

```
⇒ str.Select((char c) => c.ToString()) // "s", "t", "r", "i", "n", "g"
```

Konverze na primitivní typy (`toBoolean`, `toReal`, `toInteger`)

Hodnoty typu `String` lze převést na jiné primitivní typy pomocí operací `toBoolean():Boolean`, `toInteger():Integer` a `toReal():Real`. Operace `toBoolean` je specifikována jako ekvivalentní s porovnáním s řetězcem `'true'`.

```
⇒ 'true'.toBoolean() -- true
⇒ '1.33'.toReal() -- 1.33
⇒ '10'.toInteger() -- 10
```

Pro konverzi z řetězců na jiné hodnoty je běžné definovat statickou metoda `Parse` na cílovém typu (`double.Parse`, `int.Parse`, atd.). Metoda `bool.Parse` porovnává řetězec s hodnotami `"True"` a `"False"` a nerozlišuje velikosti písmen. Metody `double.Parse` a `int.Parse` mají několik přetížení, ve variantě s jedním argumentem závisí výsledek na nastavení národního prostředí. Aby byla konverze čísla nezávislá na národním prostředí, je nutné zadat invariantní národní prostředí. Pro konverze nezávislé na národním prostředí jsou vhodné statické metody ze třídy `System.Xml.XmlConvert`.

```
⇒ "true" == "true" // true
⇒ bool.Parse("true") // true
⇒ XmlConvert.ToBoolean("true") // true
⇒ double.Parse("1.33", CultureInfo.InvariantCulture); // 1.33D
⇒ decimal.Parse("1.33", CultureInfo.InvariantCulture); // 1.33M
⇒ XmlConvert.ToDouble("1.3") // 1.3D
⇒ XmlConvert.ToDecimal("1.3") // 1.3M
⇒ int.Parse("10", CultureInfo.InvariantCulture); // 10
⇒ XmlConvert.ToInt32("10") // 10
```

6.3.7 Logické hodnoty (`Boolean`)

Typ `Boolean` reprezentuje logickou hodnotu. Instancemi typu `Boolean` jsou hodnoty `true` a `false`, je však třeba počítat s tím, že v OCL může výraz typu `Boolean` nabývat též hodnot `null` a `invalid`.

```
⇒ true -- literál true
⇒ false -- literál false
```

V C# je `bool` (`System.Boolean`) primitivní hodnotový typ a může nabývat pouze hodnot `true` a `false`, pro reprezentaci `null` je nutné použít typ `bool?`.

Binární logické operátory (`and`, `or`, `implies`, `xor`)

Operace `and(b:Boolean):Boolean` vrací logický součin dvou hodnot. Dokáže ovšem zastavit propagaci hodnot `invalid` a `null`, neboť vrací `false`, pokud alespoň

jeden operand je `false`. Operace `or(b:Boolean):Boolean` resp. `implies(b:Boolean):Boolean` počítá logický součet resp. logickou implikaci; jejich chování pro hodnoty `null` a `invalid` je obdobné jako u operace `and`. Operace `xor(b:Boolean):Boolean` je na rozdíl od výše uvedených operací pro hodnotu `invalid` definována obvyklým způsobem. Přesné hodnoty jsou uvedeny v tabulce 6.4.

a	b	a and b	a or b	a implies b	a xor b
true	true	true	true	true	false
true	false	false	true	false	true
true	null	null	true	null	null
true	invalid	invalid	true	invalid	invalid
false	true	false	true	true	true
false	false	false	false	true	false
false	null	false	null	true	null
false	invalid	false	invalid	true	invalid
null	true	null	true	true	null
null	false	false	null	null	null
null	null	null	null	null	null
null	invalid	invalid	invalid	invalid	invalid
invalid	true	invalid	true	true	invalid
invalid	false	false	invalid	invalid	invalid
invalid	null	invalid	invalid	invalid	invalid
invalid	invalid	invalid	invalid	invalid	invalid

Tabulka 6.4: Binární logické operace v OCL

Částečnou chopnost zastavit propagaci hodnoty `invalid` reprezentovanou výjimkou, má v C# operátor `&&` resp. `||`, kde pokud první operand je `false` resp. `true`, druhý operand se nevyhodnocuje. Tyto operátory ovšem nelze použít na typu `bool?`. Operátory `&` a `|` zpracovávají hodnotu `null` stejně jako operátory v OCL, vyhodnocují ovšem narozdíl od `&&` a `||` vždy oba operandy. Pro implikace C# neobsahuje žádný operátor – vyjádříme ji znegováním levého operandu. Operátor `^` odpovídá operaci `xor`, podobný je `!=`, který se liší pro `null`. Všechny operátory jsou popsány v tabulce 6.5.

a	b	a&b	a|b	a^b	a!=b	a&&b	a||b
true	true	true	true	false	false	true	true
true	false	false	true	true	true	false	true
true	null	null	true	null	true		
true	výjimka	výjimka	výjimka	výjimka	výjimka	výjimka	true
false	true	false	true	true	true	false	true
false	false	false	false	false	false	false	false
false	null	false	null	null	true		
false	výjimka	výjimka	výjimka	výjimka	výjimka	false	výjimka
null	true	null	true	null	true		
null	false	false	null	null	true		
null	null	null	null	null	false		
null	výjimka	výjimka	výjimka	výjimka	výjimka		
výjimka	true	výjimka	výjimka	výjimka	výjimka	výjimka	výjimka
výjimka	false	výjimka	výjimka	výjimka	výjimka	výjimka	výjimka
výjimka	null	výjimka	výjimka	výjimka	výjimka		
výjimka	výjimka	výjimka	výjimka	výjimka	výjimka	výjimka	výjimka

Tabulka 6.5: Binární logické operátory v C#, včetně výjimek

Logická negace (`not`)

Logická negace `not():Boolean` je popsána v tabulce 6.6.

a	not b
true	false
false	true
null	null
invalid	invalid

Tabulka 6.6: Operace `not` v OCL

V C# použijemene operátor `!` (tabulka 6.7).

a	!a
true	false
false	true
null	null
výjímka	výjimka

Tabulka 6.7: Operátor `!` v C#

Konverze na řetězec (`toString`)

Operace `toString():String` převede hodnotu `Boolean` na řetězec, tedy pro hodnotu `true` vrátí `'true'` a pro `false` vrátí `'false'`.

```
⇒ let b:Boolean = true in b.toString() -- 'true'
```

V jazyce C# má typ `bool` metodu `ToString`, ta ovšem vrací řetězce s velkým písmenem `"True"` nebo `"False"`. Řetězce s malými písmeny dostaneme při použití metody `ToString` ze třídy `System.Xml.XmlConvert`.

```
bool b = true;
⇒ b.ToString() // "True"
⇒ XmlConvert.ToString(b) // "true"
```

6.3.8 Reálná čísla (`Real`)

Typ Real reprezetuje reálná čísla bez omezení rozsahu nebo přesnosti.

```
⇒ 3.1415 -- literál typu Real
```

V C# jsou reálná čísla omezena jak co do rozsahu tak do přesností. Primitivní typ `double` (`System.Double`) je dvojkový, 64-bitový s plovoucí řádovou čárkou[6]. Navíc umí reprezentovat speciální hodnoty jako kladné a záporné nekonečno, záporná nula a „není číslo“ (NaN). Druhá možnost je primitivní typ `decimal` (`System.Decimal`), 128-bitový s plovoucí desetinnou čárkou.

```
⇒ 3.1415D // literál typu double
⇒ 3.1415M // literál typu decimal
```

Artimetické operace (+, -, *)

Binární operátory `+(r:Real):Real`, `-(r:Real):Real`, `*(r:Real):Real` a unární operátor `-():Real` mají v jazycích OCL a C# (pro typy `double` a `decimal`) stejný význam.

Dělení (/)

Binární operátor `/(r:Real):Real` dělí reálná čísla. Pokud je hodnota parametru `0`, vrací hodnotu `invalid`.

```
⇒ 1.0/0.0 -- invalid
```

V jazyce C# může operátor `/` na typu `double` v závislosti na znaménku prvního operandu vrátit hodnoty `PositiveInfinity`, `NegativeInfinity` nebo `NaN`. U typu `decimal` dělení nulou způsobuje výjimku `DivideByZeroException`.

```
double zerod = 0; decimal zerom = 0;
⇒ 1D/zerod // PositiveInfinity
⇒ 0D/zerod // NaN
⇒ 1M/zerom // DivideByZeroException
```

[6] dle standardu IEEE 754-1985 na který se odkazuje i http://www.w3.org/TR/xmlschema-2/#double zmíněný ve specifikaci OCL

Zaokrouhlování (`floor`, `round`)

Operace `floor():Integer` vací dolní celou část reálného čísla. Operace `round():Integer` zaokrouhluje hodnotu na nejbližší celé číslo; tam, kde to není jednoznačné zaokrouhluje směrem vzhůru.

Statická metoda `Math.Floor` vrací dolní celou část, avšak jako typ `double` resp. `decimal`, takže výsledek je nutné ještě přetypovat na celé číslo. Samotné přetypování na `int` chová jinak pro záporná čísla. Pro zaokrouhlování se používá statická metoda `Math.Round`, hodnoty uprostřed ale zaokrouhluje na nejbližší sudé celé číslo. Toto chování lze změnit přidáním argumentu typu `MidpointRounding`, kerý má ovšem jen 2 možné hodnoty, `MidpointRounding.ToEven` a `MidpointRounding.AwayFromZero`, tedy zaokrouhlení na sudé (výchozí) a směrem od nuly. Jak je vidět v tabulce 6.8, obě se liší od `round` v OCL.

	OCL		C#			
d	d.round()	d.floor()	Math.Round(d)	Math.Round(d, MidpointRounding.AwayFromZero)	Math.Floor(d)	(int)d
1.5	2	1	2	2	1	1
0.7	1	0	1	1	0	0
0.5	1	0	0	1	0	0
0.3	0	0	0	0	0	0
-0.3	0	-1	0	0	-1	0
-0.5	0	-1	0	-1	-1	0
-0.7	-1	-1	-1	-1	-1	0
-1.5	-1	-2	-2	-2	-2	-1

Tabulka 6.8: Zaokrouhlování reálných čísel v C# a OCL

Extrémy (`max`, `min`)

Operace `max(r:Real):Real` a `min(r:Real):Real` vrátí větší a menší ze dvou reálných čísel.

```
⇒(1.3).max(3.6)
⇒(1.3).min(3.6)
```

V C# použijme statické metody třídy `System.Math` přetížené pro `double` nebo `decimal`.

```
⇒Math.Max(1.3, 3.6) // 3.6
⇒Math.Min(1.3, 3.6) // 1.3
⇒Math.Max(1.3M, 3.6M) // 3.6M
⇒Math.Min(1.3M, 3.6M) // 1.3M
```

Porovnávání reálných čísel (`<`, `>`, `<=`, `>=`)

Porovnávání reálných čísel (`<(r:Real):Real`, `>(r:Real):Real`, `<=(r:Real):Real` a `>=(r:Real):Real`) mají v jazycích OCL a C# stejnou sémantiku.

Konverze na řetězec (`toString`)

Operace `toString():String` převede reálné číslo na řetězec.

```
⇒ (1.3).toString() -- '1.3'
```

U typů `double` a `decimal` k tomu slouží instanční metoda `ToString()`, která převede číslo na řetězec, ovšem výsledek závisí na nastavení národního prostředí. Chceme-li konzistentní formát, jako argument metody `ToString` můžeme explicitně specifikovat neutrální kulturu, nebo použijeme statickou metodu třídy `System.Xml.XmlConvert`.

```
⇒ 1.3D.ToString(CultureInfo.InvariantCulture) // "1.3"
⇒ 1.3M.ToString(CultureInfo.InvariantCulture) // "1.3"
⇒ XmlConvert.ToString(1.3D) // "1.3"
⇒ XmlConvert.ToString(1.3M) // "1.3"
```

6.3.9 Celá čísla (`Integer`)

Jazyky UML a OCL definují typ `Integer`, jehož instancemi jsou všechna celá čísla, bez omezení velikosti. Typ `Integer` je kompatibilní s typem `Real`.

```
⇒ 100000000000000 -- literál typu Integer
```

V jazyce C# mají primitivní celočíselné typy omezenou velikost, nejběžnější je typ `int` (`System.Int32`). Tento typ má omezenou velikost na 32 bitů a chování při přetečení závsí na tom, zda se výraz nachází v kontextu `checked` (vyhození výjimky `OverflowException`) nebo `unchecked` (oříznutí na 32 bitů). Přetečení v konstantním výrazu v kontextu `checked` způsobí chybu již při překladu. Větší rozsah 64 bitů má primitivní typ `long` (`System.Int64`). Celá čísla neomezené velikosti (omezené velikostí dostupné paměti, při jejímž překročení dojde k výjimce `OutOfMemoryException`) jsou k dispozici díky typu `System.Numerics.BigInteger`. Jelikož se jedná o hodnotové typy, k reprezentaci hodnoty `null` použijeme nullable varianty těchto typů (`int?`, `long?`, `BigInteger?`).

```
⇒ 10000 // literál typu int (System.Int32)
⇒ checked(200000*200000) // Chyba překladu (přetečení)
int i = 200000;
⇒ checked(i*200000) // OverflowException (přetečení)
⇒ unchecked(200000*200000) // 1345294336 (přetečení)
⇒ checked(200000L*200000) // 40000000000 (typ long, System.Int64).
⇒ new BigInteger(200000)*new BigInteger(200000) // 40000000000
```

Aritmetické operátory (+, -, ∗)

Binární operátory +, -, ∗ a unární operátor - mají v jazycích OCL a C# stejnou sémantiku.

Dělení (`/`, `div`, `mod`)

Oba jazyky podporují též operátor dělení, v OCL dělení operátorem `/(i: Integer):Real` probíhá v reálných číslech, v C# ovšem znamená dělení celočíselné. Proto je potřeba předem alespoň jeden z operandů přetypovat na reálný typ. Celočíselný podíl a zbytek po celočíselném dělení vracejí operace `div(i:Integer): Integer` a `mod(i:Integer):Integer`, čemuž v jazyce C# slouží binární operátory `/` a `%`. Zaokrouhluje se směrem k nule. V případě dělení nulou vrací všechny tři operace hodnotu `invalid`, v C# vyvolávají výjimku `DivideByZeroException`.

```
⇒ 5/2 -- 2.5 (reálné dělení)
```

```
⇒ 5/2 // 2 (celočíselné dělení)
⇒ (double)j / 2 // 2.5 (reálné dělení)
```

Absolutní hodnota (`abs`)

Operace `abs():Integer` vrátí absolutní hodnotu celého čísla.

```
⇒ (-1).abs() -- 1
```

K získání absolutní hodnoty poskytuje BCL statickou metodu `Math.Abs` přetíženou pro primitivní typy.

```
⇒ Math.Abs(-1) // 1
```

Extrémy (`max`, `min`)

Operace `max(i:Integer):Integer` a `min(i: Integer):Integer` vrátí větší a menší ze dvou celých čísel.

```
⇒ 1.max(2) -- 2
⇒ 1.min(2) -- 1
```

V C# opět využijeme statické metody třídy `System.Math`: `Math.Max(int)` a `Math.Min(int)`.

```
⇒ Math.Max(1, 2) // 2
⇒ Math.Min(1, 2) // 1
```

Konverze na řetězce (`toString`)

```
⇒ 1.toString() -- '1'
```

Celá čísla můžeme převést na řetězec metodou `ToString`. Stejně jako u typu `Real`, jako argument metody `ToString` můžeme explicitně specifikovat neutrální kulturu, nebo použijeme statickou metodu třídy `System.Xml.XmlConvert`.

```
⇒ 1.ToString(CultureInfo.InvariantCulture) // "1"
⇒ XmlConvert.ToString(3) // "3"
```

6.3.10 Typ `UnlimitedNatural`

Typ `UnlimitedNatural` umožňuje reprezentovat nezáporná celá čísla a speciální hodnotu $*$ (zvaná „unlimited“). Využívá se například v jazyce UML pro vyjádření horní meze kardinality[7] (`Kernel::MultiplicityElement::upper`). Typ `UnlimitedNatural` je kompatibilní s typem `Integer` a předefinovává operace `+`, `*`, `/`, `div`, `mod`, `max`, `min`, `<`, `>`, `>=`, `<=`, a `toString` a specifikuje jejich chování pro hodnotu $*$. Použití hodnoty $*$ v operacích očekávajících `Real` nebo `Integer` způsobuje hodnotu `invalid`.

```
⇒ * -- literál typu UnlimitedNatural
⇒ 1.max(*) -- *
⇒ 4 + * -- *
⇒ -* -- invalid (* je neplatný Integer)
```

V C# se velmi podobně jako hodnota $*$ chová `double.PositiveInfinity`.

```
⇒ Math.Max(1D, double.PositiveInfinity) // double.PositiveInfinity
⇒ 4D + double.PositiveInfinity // double.PositiveInfinity
```

6.3.11 Kolekce

Jazyk OCL specifikuje 4 druhy kolekcí: `Sequence`, `Set`, `OrderedSet` a `Bag`, které se liší tím, zda rozlišují pořadí prvků a zda jsou povoleny vícenásobné výskyty prvku v kolekci. Společným předkem těchto čtyř typů je abstraktní druh `Collection`. Typ kolekce je určen nejen druhem, ale i typem prvků uvedeným v závorce [8]. Kolekce mohou obsahovat prvky kompatibliní s tímto typem, včetně hodnoty `null`, ne však hodnotu `invalid`. Objekty kolekcí jsou v OCL neměnné, operace pro modifikaci kolekce vždy vracejí výsledek jako novou instanci.

V jazyce C# můžeme kolekci reprezentovat polem (`T[]`). Pole mají pevnou délku, ale jejich prvky lze měnit. Implementují rozhraní `System.Collections.Generic.IEnumerable<T>`, které umožňuje postupně projít všechny prvky (například příkazem `foreach`) a které je implementováno prakticky všemi typy reprezentujícími kolekce.

Ve jmenném prostoru `System.Collections.Generic` jsou k dispozici generické kolekce jako `List<T>` a `HashSet<T>`. Tyto kolekce nejsou neměnné a umožňují přidávání a odebírání prvků. Neměnné varianty těchto kolekcí existují ve jmenném prostoru `System.Collections.Immutable`, ovšem nejsou součástí běžné instalace .NET Frameworku. Před použitím je třeba k projektu přilinkovat assembly `System.Collections.Immutable` dostupnou např. v systému NuGet[9].

Zvláštní způsob práce s kolekcemi představují metody ze statické třídy `System.Linq.Enumerable` ([FCL], dále jen LINQ), které rozšiřují rozhraní `IEnumerable<T>`.

[7] Například vlastnost `User::payments` na obrázku 2.1 má horní mez kardinality $*$.

[8] existují tedy například typy `Collection(Integer)`, `Set(OclAny)`, kolekce mohou být i vnořené, jako `Bag(Bag(Real))`

[9] https://www.nuget.org/packages/microsoft.bcl.immutable

Typické jsou pro něj metody, které vracejí nový objekt rozhraní `IEnumerable<T>`, který reprezentuje upravenou kolekci. Prvky této upravené kolekce ovšem obvykle nejsou nikde uloženy (záleží na implementaci), ale získávají se podle potřeby až při jejím procházení (volání `MoveNext`). Získat prvky jako novou kolekci můžeme metodami `ToList`, `ToHashSet`, atd.; pro neměnné kolekce obdobně `ToImmutableList`, `ToImmutableHashSet`.

6.3.11.1 Literály kolekcí

V jazyce OCL lze kolekce zapsat jako literály s uvedením druhu kolekce a seznamu jejích prvků (typ prvků může být zadán, nebo odvozen jako společný předek uvedených prvků). Do kolekce lze kromě jednotlivých prvků také vložit interval celých čísel.

```
⇒ Bag(Integer){1, 2, 3} -- literál kolekce
⇒ Sequence{1, 2..10, 100} -- kombinace prvků a intervalů
```

V jazyce C# literály kolekcí neexistují, existuje však konstruktor pole s uvedením prvků. Podobná syntaxe je možná i pro kolekce, kde jsou prvky po zavolání bezparametrického konstruktoru do kolekce přidány voláním metody `Add`. Syntaxi pro celočíselné intervaly jazyk C# nemá, lze je však vytvořit pomocí statické metody `Enumerable.Range`.

```
int[] y = new int[]{1, 2, 3}; // Konstrukce pole
List<int> x = new List<T>{1, 2, 3}; // Syntaxe konstruktoru
x = new List<int>(); // Ekvivalentní zápis
x.Add(1);x.Add(2);x.Add(3);
⇒ Enumerable.Range(2,9) // kolekce čísel 2 až 10
```

6.3.11.2 Kompatibilita kolekcí

Kolekce v OCL jsou kovariantní, tj. například `Sequence(Integer)` je kompatibilní s `Sequence(OclAny)`. Navíc jsou všechny druhy kolekcí kompatibilní s druhem `Collection`.

```
⇒ Set{1,2}.oclIsKindOf(Collection(Real)) -- true
⇒ Set{1,2}.oclIsKindOf(Sequence(Integer)) -- false
```

V C# generické třídy obecně nejsou kovariantní, pouze generická rozhraní, která mají generický parametr označen klíčovým slovem `out`. To je případ rozraní `IEnumerable<T>`, tedy `IEnumerable<string>` je kompatibilní s `IEnumerable<object>`. Pole jsou také kovariantní. Kovariance platí pouze pro referenční typy.

```
⇒ new[]{"a", "b"} is object[] // true (referenční typ)
⇒ new[]{"a", "b"} is IEnumerable<object> // true (referenční typ)
⇒ new[]{1,2} is object[] // false (hodnotový typ)
⇒ new[]{1,2} is IEnumerable<object> // false (hodnotový typ)
```

6.3.11.3 Obecná kolekce (`Collection(T)`)

Operace společné pro všechny druhy kolekcí jsou definovány na typu `Collection(T)`, se kterým jsou kompatibilní všechny druhy kolekcí. Některé operace jsou pak v kolekcích specifických druhů předefinovány, aby mohl být použit specifičtější návratový typ (např. `flatten`) nebo upřesněna definice (např. `count`).

Porovnávání na rovnost (`=`, `<>`)

Kolekce v OCL definují operátory porovnávání na rovnost (`=(c:Collection(T)):Boolean` a `<>(c:Collection(T)):Boolean`) tak, že porovnává jak druh kolekce, tak její prvky (s přihlédnutím k četnosti u `Sequence` a `Bag` a pořadí u `Sequence` a `OrderedSet`).

```
⇒ Set{1,2}=Sequence{1,2} -- false (liší se druh kolekce)
⇒ Sequence{1,2}=Sequence{1,2} -- true (druh a prvky se shodují)
⇒ Set{1,1,2}=Set{1,2} -- true (Set nerozlišuje četnost)
```

Kolekce z knihovny BCL (včetně neměnných kolekcí) nepředefinovávají metodu `Equals`, jsou tedy porovnávány reference a ne jejich obsah. Porovnávat posloupnosti prvek po prvku ovšem můžeme pomocí rozšiřující metody `System.Linq.Enumerable.SequenceEqual` a pro provnání množin typu `System.Collections.Generic.HashSet<T>` je určena metoda `System.Collections.Generic.HashSet<T>.CreateSetComparer`, která vrátí objekt typu `System.Collections.Generic.IEqualityComparer`, jehož instanční metoda `Equals` porvnává dvě množiny.

```
var seqA=new List<int>{1,2}; var seqB=new List<int>{1,2};
⇒ seqA.Equals(seqB) // false (porovnání referencí)
⇒ seqA.SequenceEqual(seqB); // true (porovnání posloupností)
var setA=new HashSet<int>{1,2}; var setB=new HashSet<int>{1,2};
var comparer = HashSet<int>.CreateSetComparer();
⇒ comparer.Equals(setA, setB); // true (porovnání množin)
```

Testování přítomnosti prvků (`includes`, `excludes`, `includesAll`, `excludesAll`, `count`)

Operace `includes(i:T):Boolean` a `excludes(i:T):Boolean` zjišťují, zda kolekce obsahuje (resp. neobsahuje) daný prvek. Tuto vlastnost lze ověřit pro více prvků najednou operacemi `includesAll(c:Collection(T)):Boolean` a `excludesAll(c:Collection(T)):Boolean`. `count(i:T):Integer` určuje četnost zadaného prvku v kolekci[10].

```
⇒ Set{1,2,3}->includes(1) -- true
⇒ Sequence{1,2}->includesAll(Sequence{1,1,1}) -- true
⇒ Set{1,1}->count(1) -- 1
```

Kolekce, které implementují rozhraní `ICollection<T>`, poskytují metodu `Contains`; její obdoba je i v LINQ.

```
⇒ new HashSet<int>{1,2,3}.Contains(1) // true (instanční metoda)
⇒ new[]{1,2,3}.Select(x=>x).Contains(1) // true (Enumerable.Contains<T>)
```

[10] pro `Set` a `OrderedSet` vrací pouze hodnoty `0` nebo `1`

Velikost kolekce (`size`, `isEmpty`, `notEmpty`)

Operace `size():Integer` spočítá počet všech prvků v kolekci, `isEmpty():Boolean` a `notEmpty():Boolean` zjišťují, zda je kolekce prázdná resp. neprázdná.

```
⇒ null->isEmpty() -- true
⇒ Set{1}->notEmpty() -- true
⇒ Set{1..10}->size() -- 10
```

Obdobou `notEmpty` a `size` jsou rozšiřující metody `Any` a `Count` z LINQ. Kolekce implementující rozhraní `ICollection<T>` poskytují `Count` také jako vlastnost.

```
⇒ new[]{1,2}.Any() // true
⇒ new[]{1,2}.Count // 2
⇒ new[]{1,2}.Count() // 2 (Enumerable.Count<T>)
```

Aritmetické operace (`max`, `min`, `sum`)

Pokud prvky kolekce podporují operaci `max(T):T`, resp. `min(T):T`, resp. `+(T):T`, můžeme zjistit součet, maximum nebo minimum ze všech prvků v kolekci operacemi `max():T`, `min():T` nebo `sum():T`..

```
⇒ Set{1,2,3}->min() -- 1
⇒ Set{1,2,3}->max() -- 3
⇒ Set{1,2,3}->sum() -- 6
```

LINQ podporuje `Sum` na primitivních typech a jejich nullable variantách. `Min` a `Max` navíc na typech implementujících `IComparable`.

```
⇒ new[]{1,2,3}.Min() // 1
⇒ new[]{1,2,3}.Max() // 3
⇒ new[]{1,2,3}.Sum() // 6
```

Kartézský součin (`product`)

Kartézský součin operací `product(c2:Collection(T2)):Set(Tuple(first:T,second:T2))` vrací vždy množinu všech dvojic (vyjádřených jako `Tuple`), kde část `first` je prvek z první kolekce a část `second` je prvek z druhé kolekce.

```
⇒ Set{1, 2}.product(Set{3,4}) /* Set{
Tuple{first:1,second:3},
Tuple{first:1,second:4},
Tuple{first:2,second:3},
Tuple{first:2,second:4}
}*/
```

V syntaxi LINQ vyjádříme kartézský součin uvedením dvou klauzulí `from`.

```
⇒ from first in new[]{1,2} from second in new[]{3,4} select new {first, second}
```

Zploštění kolekce (`flatten`)

Operace `flatten():Collection(T2)` zploští kolekci tak, aby typ prvků nebyla kolekce. Pokud je například typem kolekce `Set(Sequence(Bag(String)))`, pak je návratový typ `Collection(String)` a výsledná kolekce obsahuje všechny řetězce, které

byly zanořené v původní kolekci. To ovšem neznamená, že by výsledná kolekce nemohla jako prvky obsahovat kolekce, můžeme mit například kolekci typu `Set(OclAny)`, které obsahuje kolekce. Jelikož ale typ prvků této kolekce je `OclAny`, nedojde ke zploštění. Na druhou stranu proměnná typu `Set(OclAny)` může například obsahovat kolekci `Set(Sequence(T))`, takže zde ke zploštění dojde. Vrácená kolekce je stejného druhu jako kolekce, na které je operace volána (vnější kolekce).

```
⇒ Set{Bag{1,2}, Bag{2,3}} -- Set{1,2,3}
```

Operaci `flatten` můžeme implementovat průchodem kolekce do hloubky, přičemž prvky na nejnižší úrovni vkládáme do výsledné kolekce.

Filtrování dle typu (`selectByKind`, `selectByType`)

Do OCL verze 2.4 byly přidány operace `selectByKind` a `selectByType`, umožňující snadné filtrování prvků podle typu (jak je definováno operacemi `isKindOf` a `isTypeOf`).

```
⇒ Set{2, 3.2, '4'}.selectByKind(Real) -- Set{2, 3.2}
⇒ Set{2, 3.2, '4'}.selectByType(Real) -- Set{3.2}
```

Rozšiřující metoda `OfType` z LINQ vybere z kolekce hodnoty kompatibilní se zadaným typem.

```
⇒ new object[]{null, 2, 3.3}.OfType<double>() // 3.3 (Enumerable.OfType)
```

Konverze mezi druhy kolekcí (`oclAsSet`, `oclAsSequence`, `oclAsBag`, `oclAsOrderedSet`)

Libovolné druhy kolekcí lze mezi sebou převádět operacemi `asSet():Set(T)`, `asSequence():Sequence(T)`, `asBag():Bag(T)` a `asOrderedSet():OrderedSet(T)`. Při konverzi na kolekce s unikátními prvky jsou případné duplicitní prvky zahozeny. Při konverzi na neuspořádané kolekce je pořadí ztraceno, naopak při konverzi na uspořádané kolekce může být uspořádání libovolné.

```
⇒ Sequence{4, 3, 3, 2}.asSequence() -- Sequence{4, 3, 3, 2}
⇒ Sequence{4, 3, 3, 2}.asOrderedSet() -- OrderedSet{4, 3, 2}
⇒ Sequence{4, 3, 3, 2}.asBag() -- Bag{2,3,3,4}
⇒ Sequence{4, 3, 3, 2}.asSet() -- Set{2,3,4}
```

Duplicitní prvky v LINQ odstraníme rozšiřující metodou `Distinct`.

```
⇒ new[]{4,3,3,2}.Distinct() // 4,3,2
```

6.3.11.4 Posloupnosti (`Sequence(T)`)

Kolekce `Seqence(T)` rozlišuje pořadí prvků a umožňuje duplicity. V jazyce C# se pro uložení posloupnosti nejčastěji využívá typ `System.Collections.Generic.List<T>`, pro neměnnou sekvenci můžeme použít `System.Collections.Immutable.ImmutableList<T>`. Oba tyto typy implementují rozhraní `System.Collections.Generic.IList<T>`.

Přístup k prvkům (`at`, `first`, `last`, `subSequence`)

K prvkům posloupnosti se často přistupuje podle jejich pozice. V OCL k tomu slouží operace `at(i:Integer):T`; parametr může nabývat hodnot od `1` do délky posloupnosti. Kromě toho můžeme použít operace `first():T` a `last():T` vracející první a poslední prvek. Podposloupnost lze získat operací `subSequence(first:Integer,last:Integer):Sequence(T)`, kde `first` a `last` jsou číslované od `1` a podposloupnost zahrnuje oba konce.

```
⇒ Sequence{1,2,3,4,5}.at(4) -- 4
⇒ Sequence{1,2,3,4,5}.first() -- 1
⇒ Sequence{1,2,3,4,5}.last() -- 5
⇒ Sequence{1,2,3,4,5}.subSequence(2,4) -- Sequence{2,3,4}
```

V C# k prvku posloupnosti přistupujeme pomocí operátoru `[]`, který je definován v rozhraní `IList<T>`. Narozdíl od jazyka OCL zde indexujeme od `0`, tedy první prvek má index `0` a poslední má index `seq.Count-1`. První a poslední prvek získáme indexací nebo metodami `First` a `Last` z LINQ. Třídy `List<T>` a `ImmutableList<T>` mají metody `GetRange`, které vracejí podposloupnost zadané délky od zadaného indexu (číslováno od `0`).

```
var seq = new List<int>{1, 2, 3, 4, 5};
⇒ seq[3] // 4
⇒ seq[0] // 1
⇒ seq.First() // 1
⇒ seq[seq.Count-1] // 5
⇒ seq.Last() // 5
⇒ seq.GetRange(1, 3) // 2, 3, 4
```

Vkládání prvků (`append`, `prepend`, `insertAt`)

Do kolekce můžeme vložit prvek na začátek, na konec nebo na zadanou pozici pomocí operací `prepend(T):Sequence(T)`, `append(T):Sequence(T)` a `insertAt(index:Integer,item:T):Sequence(T)` (indexováno od `1`). Operace `including(T):Sequence(T)` je ekvivalentní operaci `append`.

```
⇒ Sequence{1,2,3}.prepend(4) -- Sequence{4,1,2,3}
⇒ Sequence{1,2,3}.append(4) -- Sequence{1,2,3,4}
⇒ Sequence{1,2,3}.insertAt(2,4) -- Sequence{1,4,2,3}
```

Kolekce `List<T>` a `ImmutableList<T>` poskytují instanční metody `Add` (vložení na konec), a `Insert` (vložení na danou pozici, indexovanou od `0`).

```
⇒ new[]{1,2,3}.ToImmutableList().Insert(0,1) // 4,1,2,3
⇒ new[]{1,2,3}.ToImmutableList().Add(4) // 1,2,3,4
⇒ new[]{1,2,3}.ToImmutableList().Insert(1,4) // 1,4,2,3
```

Vyhledání prvku (`indexOf`)

Operace `indexOf(item:T):Integer` nalezne prvek v kolekci a vrátí jeho pozici (indexováno od `1`). Zadaný prvek musí v kolekci existovat.

```
⇒ Sequence{1,2,3}.indexOf(2) -- 2
```

Instanční metoda `IndexOf` typů `List<T>` a `ImmutableList<T>` vrací pozici indexovanou od `0`, a `-1` v případě nenalezení prvku.

```
⇒ new[]{1,2,3}.ToImmutableList().IndexOf(2) // 1
```

Otočení (`reverse`)

Pořadí prvků v posloupnosti otáčí operace `reverse():Sequence(T)`

```
⇒ Sequence{1,2,3}.reverse() -- Sequence{3,2,1}
```

Metoda `Reverse` je jak instanční na `List<T>` a `ImmutableList<T>`, tak rozšiřující metoda LINQ.

```
⇒ new[]{1,2,3}.ToImmutableList().Reverse() // 3,2,1
⇒ new[]{1,2,3}.Reverse() // 3,2,1 (Enumerable.Reverse<T>)
```

Spojení (`union`)

Operace `union(s:Sequence(T)):Sequence(T)` spojí dvě posloupnosti za sebe.

```
⇒ Sequence{1,2,3}.union(Sequence{2,3,4}) -- Sequence{1,2,3,2,3,4}
```

Vložení prvků na konec posloupnosti umožňuje metoda `AddRange` tříd `List<T>` a `ImmutableList<T>`.

```
⇒ new[]{1,2,3}.ToImmutableList().AddRange(new[]{2,3,4}) //1,2,3,2,3,4
```

Odebrání všech výskytů prvku (`excluding`)

Odebrání všech výskytů prvku provedeme operací `excluding(item:T):Sequence(T)`.

```
⇒ Sequence{1,2,3,1}.excluding(1) -- Sequence{2,3}
```

Třídy `List<T>` a `ImmutableList<T>` mají metodu Remove, ta ovšem odebere pouze první výskyt prvku. Metoda `RemoveAll` odebere všechny prvky splňující danou podmínku.

```
⇒ new[]{1,2,3,1}.ToImmutableList().Remove(1) // 2,3,1
⇒ new[]{1,2,3,1}.ToImmutableList().RemoveAll(i=>i!=1) // 2,3
```

6.3.11.5 Množiny `Set(T)`

Kolekce druhu `Set` obsahuje každý prvek nejvýše jednou v neurčeném pořadí. V knihovně BCL je implementace `HashSet<T>`, neměnná varianta ze `System.Collections.Immutable` je `ImmutableHastSet<T>`. Obě třídy implementují rozhraní `ISet<T>`.

Množinové operace (`union`, `intersection`, `symmetricDifference`, `-`)

Operace `union(s:Set(T)):Set(T)`, `itersection(s:Set(T)):Set(T)`, `symmetricDifference(s:Set(T)):Set(T)`, `-(s:Set(T)):Set(T)` provádějí běžné množinové operace.

```
⇒ Set{1,2,3}.union(Set{2,4}) -- Set{1,2,3,4}
⇒ Set{1,2,3}.intersection(Set{2,4}) -- Set{2}
⇒ Set{1,2,3}.symmetricDifference(Set{2,4}) -- Set{1,3,4}
⇒ Set{1,2,3}-Set{2,4} -- Set{1,3}
```

Všechny čtyři množinové operace jsou podporovány třídou `ImmutableHashSet` jako metody `Union`, `Intersect`, `SymmetricExcept` a `Except`. Třída HashSet má podobné metody, ovšem s příponou `With`. Kromě `SymmetricExcept` jsou také jako rozšiřující metody LINQ.

```
⇒ new[]{1,2,3}.ToImmutableHashSet().Union(new[]{2,4}) //1,2,3,4
⇒ new[]{1,2,3}.ToImmutableHashSet().Intersect(new[]{2,4}) //2
⇒ new[]{1,2,3}.ToImmutableHashSet().SymmetricExcept(new[]{2,4}) //1,3,4
⇒ new[]{1,2,3}.ToImmutableHashSet().Except(new[]{2,4}) //1,3
⇒ new[]{1,2,3}.Union(new[]{2,4}) //1,2,3,4
⇒ new[]{1,2,3}.Intersect(new[]{2,4}) //2
⇒ new[]{1,2,3}.Except(new[]{2,4}) //1,3
```

Přidávání a odebírání prvků (`including`, `excluding`)

Prvky do množiny přidáme resp. je odebereme operacemi `including(item:T):Set(T)` `excluding(item:T):Set(T)`. Při pokusu přidat existující nebo odebrat neexistující prvek je vrácena původní množina.

```
⇒ Set{1,2}.including(3) -- Set{1,2,3}
⇒ Set{1,2}.including(2) -- Set{1,2}
⇒ Set{1,2}.excluding(3) -- Set{1,2}
⇒ Set{1,2}.excluding(2) -- Set{1}
```

Třídy `ImmutableHashSet<T>` i `HashSet<T>` mají metody `Add` a `Remove`.

```
⇒ new[]{1,2}.ToImmutableHashSet().Add(3) // 1,2,3
⇒ new[]{1,2}.ToImmutableHashSet().Add(2) // 1,2
⇒ new[]{1,2}.ToImmutableHashSet().Remove(3) // 1,2
⇒ new[]{1,2}.ToImmutableHashSet().Remove(2) // 1
```

6.3.11.6 Uspořádané množiny `OrderedSet(T)`

Kolekce druhu `OrderedSet` se velmi podobá kolekci `Sequence`, jen v `OrderedSet` musejí být prvky unikátní. Druh `OrderedSet` byl do jazyka OCL přidán později než ostatní a v některých částech specifikace dosud chybí[11].

Vkládání prvků (`append`, `prepend`, `insertAt`)

`OrderedSet` má pro vkládání stejné operace jako `Sequence`: `append(item:T):OrderedSet(T)`, `prepend(item:T):OrderedSet(T)` a `insertAt(index:Integer,item:T):`

[11] Např. [RTF, Issue 5971] http://www.omg.org/issues/ocl2-rtf.html#Issue5971

`OrderedSet(T)`. OrderedSet ovšem nemůže obsahovat duplicity. Specifikace je v tomto ohledu nejasná, nicméně předpokládané chování je, že v případě pokusu o vložení již existujícího prvku je vrácena původní nezměněná kolekce[12].

```
⇒ OrderedSet{1,2,3}->append(1) -- OrderedSet{1,2,3} (prvek ignorován)
```

Další operace

Operace `first():T`, `last():T`, `at(index:Integer):T`, `indexOf(item:T):Integer` se chovají stejně jako odpovídající operace kolekcí `Sequence`. Operace `subOrderedSet (first:Integer,last:Integer):OrderedSet(T)` a `reverse():OrderedSet(T)` jsou obdobou `subSequence` a `reverse`.

6.3.11.7 Multimnožiny `Bag(T)`

Posledním druhem kolekce, neuspořádané s možností vícenásobného výskytu prvků, je `Bag(T)`.

Pro implementaci můžeme použít běžný seznam (`List<T>`), pouze je potřeba dát pozor, že při porovnání nezáleží na pořadí.

Další možnost je použít `Dictionary<T,int>`, tedy pro každou hodnotu mít uloženou její četnost. Je však třeba dát pozor na to, že prvkem `Bag` může být hodnota `null`, ovšem `null` nemůže být použit jako klíč v kolekci `Dictionary`.

Přidávání prvků (`including`, `union`)

Přidávání prvků `including(item:T):Bag(T)` možňuje přidat již existující prvek (narozdíl od `Set`). Při sjedocení (`union(bag:Bag(T)):Bag(T)`) jsou sečteny četnosti prvků v obou kolekcích. Sjednocení je možné též s množinou (`union(set:Set(T)): Bag(T)`).

```
⇒ Bag{1,2}->including(2) -- Bag{1,2,2}
⇒ Bag{1,2}->union(Bag{1,2,3}) -- Bag{1,1,2,2,3}
```

Průnik (`intersection`)

Při průniku (`intersection(bag:Bag(T)):Bag(T)`) dvou kolekcí `Bag` obsahuje výsledná kolekce prvky v menší z četností v obou kolekcích. Průnik je možný též s množinou (`intersection(set:Set(T)):Bag(T)`).

```
⇒ Bag{1,1,2,2,2}->intersection(Bag{1,1,2,3}) -- Bag{1,1,2}
```

Odebírání prvků (`excluding`)

Operace `excluding(item:T):Bag(T)` odebere všechny výskyty prvku (stejně jako u `Sequence`).

```
⇒ Bag{1,1,2,2,2}->excluding(2) -- Bag{1,1}
```

[12] [RTF, Issue 14980]http://www.omg.org/issues/ocl2-rtf.html#Issue14980

6.3.11.8 Iterátory

Iterátory jsou konstrukce jazyka, která umožňuje projít prvky kolekce a vyhodnotit výraz pro každý prvek. Použití iterátoru má podobnou syntaxi jako volání operace na kolekci. Proměnné použité pro iteraci se deklarují do závorky před znak `|`.

```
⇒ Set{1,2,3}->select(x|x>1) -- použití iterační proměnné
⇒ Set{1,2,3}->select(mod(3)==0) -- implicitní iterační proměnná
```

Obecná iterace (`iterate`)

Iterátor `iterate` je nejobecnější a ve specifikaci OCL jsou pomocí něj definovány všechny ostatní iterátory[13]. Od ostatních se liší tím, že je k dispozici akumulátor, který udržuje mezivýsledek při procházení kolekce. Akumulátor má přiřazenu výchozí hodnotu a pro každý prvek kolekce je do akumulátoru přiřazena hodnota výrazu, který se může odkazovat na aktuální prvek a aktuální hodnotu akumulátoru. Výsledkem je pak hodnota akumulátoru po zpracování všech prvků.

```
⇒ Sequence{'a','b','c'}->iterate(i; r:String = '' | i+r+i) -- 'cbaabc'
```

Stejnou funkci má rozšiřující metoda `Aggregate` z LINQ.

```
⇒ new[]{"a","b","c"}.Aggregate("", (r,i)=>i+r+i) // "cbaabc"
```

Kvantifikátory (`exists`, `forAll`)

Iterátor `exists` vrací `true`, pokud se v kolekci nachází alespoň 1 prvek splňující podmínku, `forAll` vrací `true`, pokud podmínku splňují všechny prvky kolekce. U těchto dvou iterátorů je možné použít více iteračních proměnných - pak je podmínka vyzkoušena pro všechny n-tice, kde n je počet proměnných.

```
⇒ Sequence{1, 2, 3}->exists(x | x > 1) -- true
⇒ Sequence{1, 2, 3}->forAll(x | x > 1) -- false
⇒ Sequence{1, 2, 3}->forAll(x, y | x <> y) -- true (více proměnných)
```

Pro kvantifikátory je podpora přímo v CodeContracts - metody `Contract.Exists` a `Contract.ForAll`. Ekvivalentní jsou rozšiřující metody `Any` a `All` z LINQ. Všechny ovšem podporují pouze 1 iterační proměnnou.

```
⇒ Contract.Exists(new[]{1, 2, 3}, x => x > 1) // true (CodeContracts)
⇒ new[]{1, 2, 3}.Any(x => x > 1) // true (LINQ)
⇒ Contract.ForAll(new[]{1, 2, 3}, x => x > 1) // false (CodeContracts)
⇒ new[]{1, 2, 3}.All(x => x > 1) // false (LINQ)
```

Unikátní vyhodnocení (`isUnique`)

Iterátor `isUnique` zjistí, jestli výraz vrátí pro každý prvek kolekce jinou hodnotu.

```
⇒ Sequence{1, 2, 3}->isUnique(x | x.div(2)) -- false
```

[13] Iterátor `closure` je definován rekurzí, ovšem také využívá `iterate`.

Libovolný splňující prvek (`any`)

Iterátor `any` vrátí libovolný prvek kolekce, který splňuje danou podmínku (výsledek této operace tedy není jednoznačně určen).

```
⇒ Sequence{1, 2, 3}->any(x | x > 1) -- 2 nebo 3
```

Díky volné definici je možné při implementaci vrátit vždy první vyhovující prvek. K tomu slouží rozšiřující metoda `First` z LINQ.

```
⇒ new[]{1, 2, 3}.First(x => x > 1) // 2
```

Jeden splňující prvek (`one`)

Iterátor `one` zjišťuje, zda je v kolekci právě 1 prvek splňující danou podmínku.

```
⇒ Sequence{1, 2, 3}->one(x | x > 1) -- false
```

Tento iterátor můžeme implementovat jednoduše spočítáním splňujících prvků rozšiřující metodou `Count` z LINQ. Metoda `Single` vrátí prvek kolekce, pokud je právě jeden, jinak způsobí výjimku.

```
⇒ new[]{1, 2, 3}.Count(x => x > 1) == 1 // false
⇒ new[]{1, 2, 3}.Single(x => x > 1) // InvalidOperationException
```

Uzávěr (`closure`)

Iterátor `closure` vyhodnocuje tělo iterátoru, dokud přibývají nové prvky, které vkládá do výsledné kolekce. Tělo může vracet nový prvek nebo kolekci prvků. Nové prvky jsou po jednom rekurzivně zpracovány prohledáváním do hloubky.

```
⇒ Sequence{1, 2}->closure(i|(i + 3).mod(12)) -- Sequence{1, 4, 7, 10, 2, 5, 8, 11}
```

Uzávěr můžeme implementovat rekurzí.

Filtrování prvků (`select`, `reject`)

Iterátory `select` a `reject` vždy vrací kolekci stejného typu, která obsahuje pouze ty prvky ze zdrojové kolekce, které splňují resp. nesplňují určenou podmínku (výraz typu `Boolean`).

```
⇒ Set{1,2,3}->select(i|i<3) -- Set{1,2}
⇒ Set{1,2,3}->reject(i|i<3) -- Set{3}
```

Vybrat prvky z kolekce splňující danou podmínku umí rozšiřující metoda `Where` (případně klauzule `where`) z LINQ. Pro vybrání nevyhovujících prvků podmínku znegujeme operátorem `!`.

```
⇒ new[]{1, 2, 3}.Where(i => i<3) // 1,2 (Enumerable.Where<T>)
⇒ from i in new[]{1, 2, 3} where i<3 select i // 1,2 (LINQ syntaxe)
⇒ new[]{1, 2, 3}.Where(it => !(i<3)) // 3
```

Projekce (`collectNested`, `collect`)

Iterátor `collectNested` vyhodnotí výraz pro všechny prvky kolekce a vrátí kolekci výsledků. Jelikož může výraz pro různé prvky vracet stejné hodnoty, vrácená kolekce může obsahovat duplicity. Iterátor `collect` je ekvivalentní použití operace `flatten` na výsledek iterátoru `collectNested`.

```
⇒ Set{1,2,3}->collectNested(asString()) -- Bag{'1', '2', '3'}
⇒ Set{1,2,3}->collectNested(i|Set{i, i+1}) -- Bag{Set{1,2}, Set{2,3}, Set{3,4}}
⇒ Set{1,2,3}->collect(i|Set{i, i+1}) -- Bag{1,2,2,3,3,4}
```

Obdobou `collectNested` je operátor `select` v LINQ (implementován pomocí rozšiřující metody `Select`). Metoda `SelectMany` provede zploštění výsledku o právě jednu úroveň (vnitřní výraz musí vracet `IEnumerable`). V syntaxi LINQ použití `SelectMany` dosáhneme dvěma klauzulemi `from`.

```
⇒ new[]{1, 2, 3}.Select(x => x.ToString()) // "1", "2", "3"
⇒ from x in new[]{1, 2, 3} select x.ToString() // "1", "2", "3" (syntaxe LINQ)
⇒ new[]{1, 2, 3}.SelectMany(x => new[]{x, x+1}) // 1, 2, 2, 3, 3, 4
⇒ from x in new[]{1, 2, 3} from y in new[]{x, x+1} select y
    // 1, 2, 2, 3, 3, 4 (syntaxe LINQ)
```

Setřídění (`sortedBy`)

Iterátor `sortedBy`[14] vrací vždy uspořádanou variantu kolekce, s prvky setříděnými podle výrazu, který vrací hodnoty nějakého typu `V` podorujícího porovnávání operací `<(V):Boolean`.

```
⇒ Sequence{1,2,3}->orderBy(x|-x) -- Sequence{3, 2, 1}
```

Kolekci můžeme setřídit pomocí operátoru `orderby` (resp. rozšiřující metody `OrderBy`) v LINQ. K porovnání hodnot vrácených výrazem se použije `Comparer<T>.Default`, který vyžaduje, aby typ `T` implementoval rozhraní `IComparable<T>` (nebo `System.IComparable`).

```
⇒ new[]{1, 2, 3}.OrderBy(it => -it) // 3, 2, 1
⇒ from it in new[]{1, 2, 3} orderby -it select it // 3, 2, 1 (syntaxe LINQ)
```

[14] K jeho definici byla nahlášena chyba [RTF, Issue19510], http://www.omg.org/issues/ocl2-rtf.html#Issue19510

Iterátor	Typ vnitřního výrazu	Typ zdrojové kolekce	Typ výsledku
exists, forAll	Boolean	všechny	Boolean
isUnique	V	všechny	Boolean
any	Boolean	všechny	T
one	Boolean	všechny	Boolean
closure	OclAny	Sequence(T)	OrderedSet(T)
		Set(T)	Set(T)
		Bag(T)	Set(T)
		OrderedSet(T)	OrderedSet(T)
select, reject	Boolean	Sequence(T)	Sequence(T)
		Set(T)	Set(T)
		Bag(T)	Bag(T)
		OrderedSet(T)	OrderedSet(T)
collectNested	V	Sequence(T)	Sequence(V)
		Set(T)	Bag(V)
		Bag(T)	Bag(V)
		OrderedSet(T)	Sequence(V)
collect	V	Sequence(T)	Sequence(U)
		Set(T)	Bag(U)
		Bag(T)	Bag(U)
		OrderedSet(T)	Sequence(U)
sortedBy	V	Sequence(T)	Sequence(T)
		Set(T)	OrderedSet(T)
		Bag(T)	Sequence(T)
		OrderedSet(T)	OrderedSet(T)

Tabulka 6.9: Iterátory v OCL a související typy

7. Implementovaný překlad

V rámci této práce byl implementován překlad integritních omezení z jazyka OCL do jazyka C#. Překlad podporuje všechny druhy integritních omezení dostupných v jazyce OCL.

7.1 Možnosti přístupu

Jak bylo ukázáno v předchozí kapitole, sémantika operací standardní knihovny a kompatibilita typů se v OCL a C# liší. To znamená, že korektní implementace může vést na poměrně složitá vyjádření v cílovém jazyce, ovšem pokud integritní omezení píše programátor zvyklý na sémantiku cílového jazyka, nemusí pro něj dodržení sémantiky zdrojového jazyka představovat užitek. Zvážili jsme proto tyto možnosti:

- Přizpůsobení sémantiky cílovému jazyku:
  - vygenerovaný kód kontraktů je čitelnější,
  - mezi zdrojovým a cílovým výrazem je na první pohled jasná korespondence,
  - snadnější statická analýza,
  - odolnost proti změnám ve specifikaci jazyka OCL.
- Zachování sémantiky zdrojového jazyka:
  - chování kontraktu odpovídá specifikaci,
  - je zde prostor pro pozdější vylepšení: lze detekovat případy, kdy se rozdíly nemohou projevit, v těchto případech provést zjednodušení a tím získat výhody obou přístupů.

Pro tuto práci jsem volil druhý přístup. Aby bylo dosaženo sémantiky jazyka OCL, nejsou ve výrazech používány běžné typy jazyka C# (tím jsou myšleny primitivní typy a typy definované uživatelem), ale je vytvožena knihovna *OclRuntime*, ve které jsou typy, které hodnoty běžných typů zabalují a definují na nich operace podle specifikace jazyka OCL. Aby nedocházelo ke konfliktu jmen, jako například `Exolutio.OclRuntime.String` a `System.String`, mají všechny typy ve jmenném prostoru `Exolutio.OclRuntime` prefix `Ocl`.

Implementovaný překlad podporuje několik nastavení, díky kterým lze některé výrazy mírně zjednodušit.

7.2 Výrazy jazyka OCL

7.2.1 Typy

V přeložených výrazech se tedy vyskytují dva druhy typů:

- *Zabalený typ* (wrapped) z knihovny OclRuntime používaný ve výrazech přeložených z jazyka OCL
- *Rozbalený typ* (unwrapped) je využit v deklaracích vlastností a operací. U hodnotových typů navíc rozlišujeme, zda je povolena hodnota `null`.

Přehled mapování všech typů z OCL na typy v C# ukazuje tabulka 7.1.

Typ v OCL	Typ v C#	Nullable typ v C#	Zabalený typ v C#
`Integer`	`int`	`int?`	`OclInteger`
`Boolean`	`bool`	`bool?`	`OclBoolean`
`Real`	`double`	`double?`	`OclReal`
`String`	`string`	`string`	`OclString`
`UnlimitedNatural`			`OclUnlimitedNatural`
`Collection(T)`	`IEnumerable<T>`		`OclCollection`
`Bag(T)`	`IEnumerable<T>`		`OclBag`
`Set(T)`	`IEnumerable<T>`		`OclSet`
`OrderedSet(T)`	`IEnumerable<T>`		`OclOrderedSet`
`Sequence(T)`	`IEnumerable<T>`		`OclSequence`
třída `T`	`T`	`T`	`OclObject`
datový typ `T`	`T`	`T?`	`OclObject`
výčtový typ `T`	`T`	`T?`	`OclEnum<T>`
rozhraní `T`	`T`	`T`	`OclObject`
`OclAny`			`OclAny`
`OclVoid`	`void`		`T : OclAny`
`OclInvalid`			`T : OclAny`
`Tuple`			`OclTuple`

Tabulka 7.1: Překlad typů z OCL do C#

U kolekcí, objektů a n-tic není mezi typem v OCL a zabaleným typem v C# vzájemně jednoznačná korespondence, a to z důvodu zvláštních pravidel jejich kompatibility. Více různých typů v OCL je proto přeloženo na jeden typ v C#, takže jsou všechny navzájem kompatibilní. Toto uvolnění pravidel nevadí, neboť případnou chybu v kompatibilitě u výrazů zachytí již parser. Tam, kde je potřeba kompletní informace o typu, je u objektů typ určen typem zabalené hodnoty, u kolekcí je typ prvků specifikován při konstrukci kolekce a držen v datové položce. U n-tic (`Tuple`) je typ určen typy a názvy jejích částí.

Protože `OclVoid` a `OclInvalid` jsou kompatibilní se všemi typy v OCL, mají metody vracející tyto tyy generický parametr, který je na místě volání specifikován tak, aby byla návratová hodnota kompatibilní s typem, do kterého je přiřazována.

Při přístupu k nezabalené hodnotě z výrazu OCL a naopak, při použití hodnoty výrazu OCL tam, kde je očekáván nezabalený typ, je potřeba hodnotu zkonvertovat. K tomu slouží konverzní metody v knihovně OclRuntime, vypsané ve výpisu 7.1. Při rozbalování kolekcí jsou použity instanční metody, neboť rozbalovaná kolekce nemůže být `null`. U primitivních a výčtových typů je využito přetížení konverzního operátoru.

Výpis 7.1 Konverze zabalených a nezabalených typů v C#

```
// Konverze na zabalený typ
var i = Ocl.Integer(3); // Primitivní typy
var ic Ocl.IntegerSet(new[]{1,3}); // a jejich kolekce
var c = Ocl.Object(book); // Třídy,
var d = Ocl.Object(date); // hodnotové typy,
var f = Ocl.Object(searchable); // rozhraní
var oc = Ocl.ObjectSequence(new []{book});// a jejich kolekce
var e = Ocl.Enum<UnitStatus>(UnitStatus.Present); // Výčtové typy
var ec = Ocl.EnumBag(new[]{UnitStatus.Present}); // a jejich kolekce
// Konverze ze zabaleného typu (rozbalení)
⇒ (int)i // Primitivní typy
⇒ (int?)i // Primitivní hodnotové typy včetně null
⇒ ic.GetIntegers() // Kolekce primitivních typů
⇒ OclObject.Get<T>(o) // Třídy a rozhraní
⇒ oc.GetObjects<T>() // Kolekce tříd a rozhraní
⇒ OclObject.GetValue<T>(o) // Hodnotové typy bez null
⇒ OclObject.GetNullable<T>(o) // Hodnotové typy včetně null
⇒ oc.GetValues<T>() // Kolekce hodnotových typů
⇒ (UnitStatus)e // Výčtové typy bez null
⇒ (UnitStatus?)e // Výčtové včetně null
⇒ ec.GetEnums<T>() // Kolekce výčtových typů
```

7.2.2 Operace standardní knihovny

Většina operací standardní knihovny je implementována v knihovně OclRuntime jako instanční metody příslušných typů. Metody mají název ze standardní knihovny převedený na konvence jazyka C#, u některých metod jako `flatten` nebo `toString`, kde by došlo ke kolizi s metodou s jinou návratovou hodnotou, je použito jiné jméno, např. `FlattenToSequence`, `ToOclString`. Operátory v OCL jako `+`, `*`, `-`, `/` jsou přeloženy jako operátory. Operace, které zpracovávají hodnotu `null`, jsou implementovány jako statické metody třídy `Ocl`, u těch, které navíc také zpracovávají hodnotu `invalid`, jsou argumenty předány jako lambda výrazy, v nastavení překladu však toto lze vypnout. Operace, které přidávají nové prky do kolekcí, jako argument specifikují typ prvků výsledné kolekce. Operace, které vracejí prvky z kolekcí, mají generický parametr určující typ návratové hodnoty.

```
⇒ Ocl.Integer(1) + Ocl.Integer(2) // Operátor
⇒ coll.FlattenToSequence() // Instanční metoda
```

```
⇒ Ocl.Not(Ocl.True) // Metoda zpracovávající hodnoty null
⇒ Ocl.And(() => Ocl.True, () => Ocl.False)// Metoda zpracovávající hodnoty invalid
```

7.2.3 Iterátory standardní knihovny

Iterátory jsou implementovány jako instanční metody kolekcí. Generický parametr udává typ prvků kolekce, tělo iterátoru je lambda výraz s jedním parametrem toho typu. Iterátory `collectNested`, `isUnique` a `sortedBy` mají navíc jako generický parametr návratový typ lambda výrazu, u iterátorů `collect` a `collectNested` je jako první parametr metody předán typ prvků výsledné kolekce. Iterátory `forAll` a `exists` jsou přetíženy pro jendu až tři iterační proměnné.

```
⇒ coll.CollectNested<OclObject>(OclString.Type,  // Iterátor
    unit=>unit.Get<Unit>().OclName)
⇒ coll.ForAll<OclInteger>((x, y) => x == y) // Iterátor s více proměnnými
```

7.2.4 Konstrukce `let` a `if-then-else-endif`

Konstrukce `let` je implementována jak je popsáno v sekci 6.2.4, zapnout zpracování hodnoty `invalid` je možné ve volbách překladu. Deklarovaná proměnná je vždy zabaleného typu.

Výraz `if` je přeložen jako podmínkový operátor `?:`, kde podmínka je rozbalena na typ `bool`, další dva operandy a tedy i celý výraz jsou vždy zabaleného typu.

```
⇒ Ocl.Let<OclInteger, OclInteger>(
    Ocl.Integer(1), i => i + Ocl.Integer(1)) // Konstrukce let
⇒ (bool)CalendarDate.OclIsLeapYear(oclY) ?
    Ocl.Integer(29) : Ocl.Integer(28) // Konstrukce if
```

7.2.5 Literály

Literály primitivních typů `Integer`, `Real` a `String` jsou přeloženy jako nezabalené literály typů `int`, `double` a `string`. Literály typu `Boolean` jsou přeloženy jako již zabalené datové položky `Ocl.True` a `Ocl.False`. Literál typu `UnlimitedNatural` je `Ocl.Unlimited`.

Literály `null` a `invalid` musejí být kompatibilní se všemi typy, proto jsou přeloženy jako volání generické metody, kde typový parameter je typ, do kterého je literál přiřazen. Tento přístup je využit i případě, že část výrazu obsahuje chybu nebo ji nelze přeložit. Pak je nahrazena voláním generické metody `Ocl.Error`, která vyvolá výjimku.

```
⇒ Ocl.Null<OclObject>() // null
⇒ Ocl.Invalid<OclObject>() // OclInvalidValueException
⇒ Ocl.Error<OclObject>() // OclErrorException
```

Literály kolekcí a n-tic jsou tvořeny voláním konstruktoru typu kolekce nebo metody `Ocl.Tuple`, jako parametry jsou jednotlivé části literálů a informace o jejich typu.

```
⇒ new OclSet(OclInteger.Type, // Literál kolekce
    Ocl.Integer(1), Ocl.Range(Ocl.Integer(3), Ocl.Integer(10)))
⇒ Ocl.Tuple(OclTuple.Part("x", OclInteger.Type, Ocl.Integer(1))) // Literál n-tice
```

7.2.6 Předchozí hodnoty

Pokud je operace nebo přístup k vlastnosti označený modifikátorem `@pre`, je celá tato část výrazu vyhodnocena v `Contract.OldValue`. Všechny další přístupy v podvýrazech musejí také mít modifikátor `@pre`, nesmí obsahovat odkaz na proměnnou `result` a na lokální proměnné deklarované vně takového výrazu.

Tato omezení jsou ve skutečnosti přísnější, než je nutné. Například integritní omezení

```
context User::return(u:Unit, date:CalendarDate)
post: self.Loan@pre->any(x|x.end@pre = null).end = date
```

by bylo možné přeložit jako

```
Contract.Ensures((bool)(
  OclObject.Get<Loan>(Contract.OldValue( this.OclLoan.Any<OclObject>(
    x => OclObject.Get<Loan>(x).OclEnd == Ocl.Null<OclObject>()
  ))).OclEnd == date
));
```

Zde by se využilo toho, že ač proměnná `x` není deklarovaná v podvýrazu nějakého přístupu s modifikátorem `@pre`, všechny přístupy k ní jej mají, a kolekce, přes kterou se iteruje, je také předchozí hodnota. Potom lze celý iterátor vyhodnotit v `Contract.OldValue`. Tělo iterátoru v příkladu ovšem obsahuje i volání operace `=`, které není označeno modifikátorem `@pre`, příklad se tedy zároveň spoléhá na to, že operace `=` vrací pro porovnávané objekty vždy stejný výsledek nezávisle na tom, kdy je vyhodnocena.

7.3 Typy z modelu UML

Podporováno je generování tříd, rozhraní, datových typů a výčtových typů. Každý typ je vygenerován do samostatného souboru. Ten začíná hlavičkou obsahující direktivy `using` a deklaraci jmenného prostoru, jehož název je zadán v nastavení překladu. Následuje samotná definice typu. U rozhraní a tříd obsahujících abstraktní metody je navíc přidána abstraktní třída s příponou `Contracts`, která obsahuje těla metod s jejich integritními omezeními. U rozhraní tato třída musí obsahovat deklarace všech metod a vlastností rozhraní i jeho předků.

Třídy, rozhraní a datové typy, které mají invarianty, mají metodu `Invariants`. U rozhraní se nachází v pomocné abstraktní třídě. Invarianty jsou volání metody `Contract.Invariant`, jako argument je přeložený výraz rozbalený na typ `bool`.

7.4 Vlastnosti

Každá vlastnost může mít nejvýše jedno z integritních omezení `derive`, `init` nebo `def`.

- Pro abstraktní vlastnosti tříd a vlastnosti rozhraní je vygenerována pouze deklarace getteru.
- Vlastnosti tříd a hodnotových typů, které mají integritní omezení `derived`, mají pouze getter, který vrací příslušný výraz.
- Vlastnosti definované pomocí konstrukce `def` mají zabalený typ a k jejich názvu je přidána předpona `Ocl`.
- Vlastnosti, které nemají omezení `derived` ani `def`, a které nejpředefinovávají zděděnou vlastnost, ukládají hodnotu v privátní datové položce, nebo pomocí automatické implementace (pokud je povoleno v nastavení překladu).
- Ke vlastnostem tříd a hodnotových typů, které nejsou definované pomocí `def` a které jsou použité v nějakém výrazu jazyka OCL, je vygenerována vlastnost s předponou `Ocl`, která vrací zabalenou hodnotu. Díky tomu není nutné hodnoty konvertovat na všech místech, kde je k ním přistupováno.
- Vlastnosti, které mají omezení `init`, jsou na specifikovanou hodnotu inicializované buď v deklaraci privátní datové položky, nebo v konstruktoru (pokud privátní datová položka není vytvořena).
- Vlastnosti hodnotových typů jsou inicializovány v jeho konstruktoru.

7.5 Operace

Operace může mít libovolný počet integritních omezení `post` a `pre`, a nejvýše jedno z `body` a `def`.

- Pro abstraktní operace tříd a operace rozhraní je vygenerována pouze deklarace.
- Operace tříd a hodnotových typů, které mají integritní omezení `body`, vracejí příslušný výraz. Pokud mají výstupní nebo vstupně-výstupní parametry, je

hodnota výrazu uložena do proměnné typu `OclTuple`, z níž jsou pak jednotlivé části přiřazeny do výstupních a vstupně-výstupních parametrů.

- Operace definované pomocí konstrukce `def` mají zabalené typy parametrů a návratové hodnoty a k jejich názvu je přidána předpona `Ocl`.
- Operace, které nemají omezení `derived` ani `def`, vyvolávají výjimku `NotImplementedException`.
- Návratová hodnota (proměnná `result`) se získá voláním `Contract.Result`, ale pokud má operace výstupní nebo vstupně-výstupní parametry, pak je hodnota proměnné `result` sestavena jako `Tuple` z `Contract.Result` a hodnot těchto parametrů. K výstupním parametrům se přistupuje pomocí `Contract.ValueAtReturn`.
- K parametrům metody ve výstupních podmínkách se přistupuje pomocí `Contract.OldValue`. To platí i pro vstupní parametry, protože v C# jsou parametry proměnné, které se mohou měnit. Pomocí volby překladu lze povolit vynechání `Contract.OldValue` pro přístup ke vstupním parametrům.
- Omezení `pre` a `post` jsou přeložena na `Contract.Requires` a `Contract.Ensures` v těle metody. Pro abstraktní operace a operace rozhraní jsou kontrakty v pomocné abstraktní třídě. V nastavení překladu lze povolit generování výstupních podmínek `EnsuresOnThrow`. Tyto podmínky jsou shodné s podmínkami `Contract.Ensures`, jen jsou v nich všechny přístupy k proměnné `result` nahrazeny konstantou `invalid`.
- K operacím, které nejsou definované pomocí `def` a které jsou použité ve nějakém výrazu jazyka OCL, je vygenerována pomocná metoda s předponou `Ocl`, která má zabalené typy parametrů a návratové hodnoty. Tato obalovací metoda provede rozbalení argumentů, zavolá původní metodu a návratovou hodnotu zabalí a vrátí. Pokud má metoda výstupní nebo vstupně-výstupní parametry, pak obalovací metoda nejprve deklaruje lokální proměnné, které jsou použity pro uložení hodnot těch parametrů. Poté zavolá obalenou metodu a vrátí n-tici, skládající se z návratové hodnoty a hodnot výstupních a vstupně-výstupních parametrů. Obalovací metoda je pak používána ve výrazech jazyka OCL. Výpis 7.2 ukazuje, jak jsou výstupní a vstupně výstupní parametry zpracovány pro operaci `availableUnit` z příkladu na obrázku 2.1. Pro metody rozhraní jsou jejich obalovací metody vygenerovány v pomocné abstraktní třídě.

Výpis 7.2 Příklad metody a obalovací metody pro operaci s výstupními parametry.

```
[Pure]
public virtual Unit AvailableUnit(
  Title t,
  ref CalendarDate date,
  out UnitStatus status)
{
  Contract.Ensures((bool)(OclObject.Get<Unit>(Ocl.Tuple(
    OclTuple.Part("result", OclObjectType.Get<Unit>(),
      Ocl.Object(Contract.Result<Unit>())),
    OclTuple.Part("date", OclObjectType.Get<CalendarDate>(), Ocl.Object(date)),
    OclTuple.Part("status", OclEnumType.Get<UnitStatus>(),
      Ocl.Enum<UnitStatus>(Contract.ValueAtReturn(out status)))
  ).Get<OclObject>("result")).OclLibrary == Ocl.Object(this)));
  ...
}
[Pure]
public OclTuple OclAvailableUnit(OclObject oclT, OclObject oclDate) {
  CalendarDate date = OclObject.GetValue<CalendarDate>(oclDate);
  UnitStatus status;
  OclObject result = Ocl.Object(this.AvailableUnit(
    OclObject.Get<Title>(oclT), ref date, out status));
  return Ocl.Tuple(
    OclTuple.Part("result", OclObjectType.Get<Unit>(), Ocl.Object(result)),
    OclTuple.Part("date", OclObjectType.Get<CalendarDate>(), Ocl.Object(date)),
    OclTuple.Part("status", OclEnumType.Get<UnitStatus>(),
      Ocl.Enum<UnitStatus>(status))
  );
}
```

7.6 Kardinality

Vlastnosti a parametry v UML mají kardinalitu, která sestává z dolní a horní meze na počet hodnot. Při překladu rozlišujeme tři případy:

- Obě meze jsou 1 (`[1]`), pak hodnota vlastnosti či parametru nesmí být `null`. Pokud je jejich typ v C# referenční, můžeme přidat kontrakt, který toto kontroluje.
- Horní mez je 1 a dolní 0 (`[0..1]`), pak hodnota může být `null`. U hodnotových typů je použita nullable varianta (třetí sloupec v 7.1).
- Horní mez je větší než 1, pak se jedná o kolekci hodnot. Rozbalený typ je pro všechny druhy kolekcí `IEnumerable<T>`, ale při zabalení je druh kolekce určen podle toho, zda má vlastnost nebo parametr příznaky `ordered` a `unique` (tabulka 7.2). Kolekce nesmí být `null` a pokud je dolní mez větší než `0` nebo horní mez menší než $*$, pak můžeme přidat kontrakt, který porovnává délku kolekce s těmito mezemi.

Generování kontraktů podle kardinality lze povolit ve volbách překladu. U vlastností jsou přidány jako invarianty (příklad ve výpisu 7.3), u parametrů (a návra-

tové hodnoty) jako vstupní a výstupní podmínky operace.

{ordered}	{unique}	Typ v OCL	Typ v C#	Zabalený typ v C#
ne	ne	`Bag(T)`	`IEnumerable<T>`	`OclBag`
ano	ne	`Sequence(T)`	`IEnumerable<T>`	`OclSequence`
ne	ano	`Set(T)`	`IEnumerable<T>`	`OclSet`
ano	ano	`OrderedSet(T)`	`IEnumerable<T>`	`OclOrderedSet`

Tabulka 7.2: Typy v OCL pro vícečetné vlastnosti a parametry

Výpis 7.3 Integritní omezení odvozené z kardinality vlastnosti

```
class User{
  ...
  [ContracInvariantMethod] public void Invarinants(){
    Contract.Invariant(names!=null && names.Count()>=1 && names.Count()<=3);
  }
}
```

7.7 Příklad vygenerovaného kódu

Výpis 7.4 ukazuje vygenerovaný kód pro soubor `CD.cs`, obsahující třídu `CD` z příkladu na diagramu 2.1.

Výpis 7.4 Vygenerovaný kód pro třídu `Category` (formátování upraveno)

```
using Exolutio.OclRuntime;
using System;
using System.Collections.Generic;
using System.Diagnostics.Contracts;
public class CD : Title {
  public virtual string Ean { get; set; }
  public OclString OclEan {
    get { return Ocl.String(this.Ean); }
  }
  public virtual IEnumerable<string> Tracks { get; set; }
  public OclSet OclTracks {
    get { return Ocl.StringSet(this.Tracks); }
  }
  [Pure]
  public override bool Matches(string query) {
    Contract.Requires(query != null);
    return (bool)Ocl.Or(
     () => this.OclName.IndexOf(Ocl.String(query)) > Ocl.Integer(0),
     () => this.OclTracks.Exists<OclString>(
       t => t.IndexOf(Ocl.String(query)) > Ocl.Integer(0)));
  }
  [ContractInvariantMethod]
  private void Invariants() {
    Contract.Invariant((bool)(this.OclId == Ocl.String("CD") + this.OclEan));
    Contract.Invariant(this.Ean != null);
    Contract.Invariant(this.Tracks != null && this.Tracks.Count() >= 1);
  }
}
```

8. Implementace

Překlad integritních omezení popsaný v předchozí kapitole byl implementován do programu eXolutio. To zahrnuje:

- drobné rozšíření uživatelského rozhraní,
- doplnění existujícího UML modelu,
- doplnění existujícího parseru jazyka OCL,
- implementaci samotného překladu a generování kódu,
- běhovou knihovnu OclRuntime, obsahující implementaci standardní knihovny a pomocné třídy.

8.1 Doplnění programu eXolutio

Hlavním tématem programu eXolutio je práce technologiemi XML. Obsahuje ovšem také parser jazyka OCL a s ním i část implementace metamodelu UML. Tyto části proto byly použity v této práci, ale vyžádaly si úpravy.

8.1.1 Model

Model v programu eXolutio může být nezávislý na platformě (PIM) nebo specifický pro platformu (PSM). PSM se je svázán s technologiemi XML a v této práci není použit.

Jako model byl využit PIM a pro potřeby této práce rozšířen o:

- rozhraní a hodnotové typy,
- výčtové typy,
- příznaky unique a ordered,
- příznaky abstract a static,
- rozlišení směrů u parametrů operací.

Úpravy se týkaly především jmenných prostorů `Exolutio.Model.PIM` a `Exolutio.Model.OCL.Bridge`.

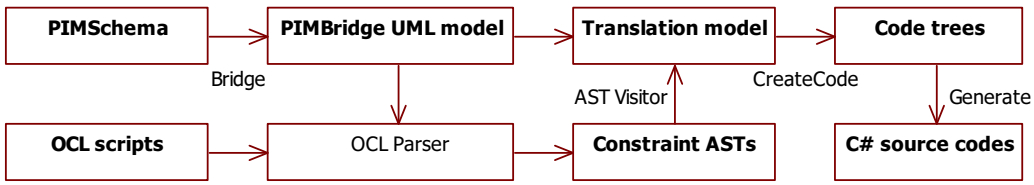
8.1.2 Parser

Parser jazyka OCL v eXolutio již podporoval kompletní syntaxi jazyka OCL, nebylo ale dokončeno vytváření abstraktního syntaktického stromu pro všechny druhy integritních omezení. Do parseru jazyka OCL (`Exolutio.Model.OCL.Compiler`) byla doplněna podpora konstrukcí `pre`, `post` a `body`. Výrazné úpravy si vynutila podpora konstrukce `def`.

8.1.3 Uživatelské rozhraní

Program eXolutio má grafické uživatelské rozhraní založené na frameworku WPF. V této práci byla využita nástrojová lišta, editor diagramu tříd, editor OCL skriptů a karty pro vygenerované soubory. Do nástrojové lišty byla přidána nová tlačítka na generování kódu (implementace je v `Exolutio.View.Commands.CodeContracts`). Generované soubory se v původním programu otevíraly v samostatných oknech, což je při větším počtu generovaných kódů velmi nepřehledné, proto byla přidána podpora zobrazení v hlavním okně. Vzhledem k tomu, že byly rozšířeny možnosti modelu, by bylo vhodné rozšířit i editor diagramu tříd, aby umožnil nové elementy vytvářet a upravovat, to ale nebylo provedeno.

8.2 Implementace překladu



Obrázek 8.1: Schéma implementace překladu

Obrázek 8.1 znázorňuje základní schéma překladu od modelu PIM a skriptů jazyka OCL až po vygenerovaný kód. Nejprve je na základě modelu (`PIMSchema`) vytvořen model `PIMBridge`, který je poté použit při parsování skriptů v jazyce OCL. Potud se jedná o část, která v programu eXolutio již byla. Následuji hlavní část implementace se nachází v projektu `CodeContracts`, ve jmenném prostoru `Exolutio.CodeContracts`.

8.2.1 Model

Veřejné API překladu tvoří statická třída `Translations`, která jako vytvoří objekt `Translation`, který řídí překlad.

Podle modelu `PIMBridge` se vytvoří nový model, který obsahuje odkazy na původní model a navíc udržuje rozličné atributy použité při překladu, a části generovaného kódu. Stejná věc je poté provedena pro integritní omezení a všechny uzly syntaktického stromu výrazů v nich. Integritní omezení jsou doplněna k příslušným typům, operacím a vlastnostem.

Takto vytvořený model je poté třikrát procházen do hloubky. V prvním průchodu (`SolveTypeAttributes`) jsou vypočteny hodnoty atributů u typů, operací a vlastností. Ve druhé fázi `SolveContraintAtributes` jsou vyřešeny atributy integritních omezení a všech výrazů. V poslední fázi `CreateFiles` je vytvořena struktura zdrojového kódu a vygenerován kód.

8.2.1.1 Přehled elementů modelu

Typy (`IType`)

Typ může být `Class`, `Enumeration`, `DataType` nebo `Interface`. Každý typ generuje jeden soubor (`SourceFile`), který obsahuje kód typu a případně metodu `Invariants` a pomocnou třídu `ContractClass`.

Integritní omezení (`IConstraint`)

Integritní omezení generuje výraz, který je ověřuje (`ConstraintCode`). Integritní omezení jsou Invariant, `OperationBody`, `OperationDefinition`, `OperationPostcondition`, `OperationPrecondition`, `PropertyDefinition`, `PropertyDerivedValue` a `PropertyInitialization`.

Výrazy (`IOclExpression`, `ExpressionContext`)

`ExpressionContext` představuje místo, kde se vyskytuje výraz jazyka OCL, tedy v integritním omezení a uzlech AST, které nejsou listy. Jeho atributy určují, jakého typu má výraz být (`ExpectedType`), zda má být hodnota zabalena (`IsWrapped`) a zda má být výraz vyhodnocen ve volání `Contract.OldValue` (`NeedOldValue`).

V každém `ExpressionContext` pak je výraz `IOclExpression`, který odpovídá uzlu z AST. Její atributy určují, zda bylo možné výraz přeložit, jestli je jeho hodnota zabalená a jestli má být vyhodnocena v `Contract.OldValue`. Pokud se hodnoty atributů výrazu neshodují s atributy kontextu, pak je přidán kód na zabalení nebo rozbalení hodnoty nebo volání `Contract.OldValue`. V případě, že výraz nelze přeložit, je místo něj generováno volání metody `Ocl.Error`.

Vlastnosti

Vlastnosti z modelu (`Property`) generují části kódu, které jsou vloženy do těla typu, konstruktoru, pomocné abstraktní třídy nebo metody `Invariants`. Při použití vlastnosti ve výrazu (`PropertyCallExp`) je vytvořen pomocný výraz `Property.PropertyReference`, která generuje kód přístupu k vlastnosti. `PropertyCallExp` může být použito pro přístup do n-tic (`Tuple`), pak je kód generován pomocným výrazem `TuplePartReference`.

Operace (`IOperationImplementation`)

Operace z modelu (`Operation`) také generují části kódu, které jsou vloženy do těla typu nebo pomocné abstraktní třídy. Při použití operace z výrazu (`OperationCallExp`) je vytvořen pomocný výraz `Operation.ModelOperationCall`, který generuje kód volání. Implementace operací standardní knihovny se nachází ve jmenném prostoru `Exolutio.CodeContracts.Translation.Implementation`. Třída `LibraryImplementation` obsahuje mapování jednotlivých operací standardní knihovny na jejich implementace.

Iterátory (`IIteratorImplementation`)

Iterátory použité v `IteratorCallExp` jsou také generovány pomocí implementací specifikovaných v `LibraryImplementation`.

Proměnné (`IVariable`)

Proměnné mohou být parametry operací z modelu. Parametry (`Parameter`) generují kód pro deklaraci v metodě a obalovací metodě, v případě výstupních a vstupně výstupních parametrů navíc pomocné proměnné a kód tvoření části výsledné `Tuple` v obalovací metodě.

Lokální proměnné (`Variable`) mohou být deklarovány v konstrukci `let` a iterátorech.

Při použití ve výrazu (`VariableExp`) je kód generován pomocí pomocného výrazu `Parameter.ParameterReference` pro parametry, `VariableVariableReference` pro lokální proměnné, `ResultVariable` pro proměnnou `result` a `SelfVariable` pro proměnnou `self`.

Přidělování jmen

Jména elementů v PIM modelu nemusejí být validní identifikátory jazyka C#, z každého jména jsou proto odstraněny zakázané znaky a je upravena velikost

písmen, aby odpovídala zvyklostem jazyka C#. Pokud vzniklá jména navzájem kolidují, jsou ke jménu připojena postupně rostoucí čísla.

Pomocné třídy

Třídy `ExpressionUtils`, `TypeUtils` a `MultiplicityUtils`, poskytují metody pro často opakované vytváření kódu.

8.2.2 Generování zdrojových kódů

Pro generování výsledného zdrojového kódu v jazyce C# jsme zvážili několik existujících technologií.

CodeDom[1]

CodeDom je součástí Microsoft .NET Frameworku. Je to knihovna tříd sloužících pro generování kódů v jazycích pro .NET, konkrétně C# a VisualBasic. Kód je stavěn do podoby acyklického grafu, jehož vrcholy představují jednotlivé typy, jejich členy, příkazy a výrazy. Takto vytvořená struktura je nezávislá na cílovém jazyku a proto nepodporuje konstrukce specifické pro jazyk C#: lambda výrazy, syntaxi LINQ, unární operátory. CodeDom ani nemá mechanizmy rozšíření, které by umožnily chybějící syntaktické konstrukce doplnit. Příklad možného použití je ve 8.1.

Výpis 8.1 Příklad generování kódu pomocí CodeDom

```
// Metoda Contract.Requires
var contract = new CodeTypeReferenceExpression("Contract");
var contractRequires = new CodeMethodReferenceExpression(contract, "Requires");
// Výraz i > 1
var i = new CodeVariableReferenceExpression("i");
var one = new CodePrimitiveExpression(1);
var expr = new CodeBinaryOperatorExpression(
    i, CodeBinaryOperatorType.GreaterThan, one);
// Volání metody
var call = new CodeMethodInvokeExpression(contractRequires, expr);
// Příkaz
var stmt = new CodeExpressionStatement(call);
// Generování kódu
CodeDomProvider cp = CodeDomProvider.CreateProvider("CSharp");
var writer = new System.IO.StringWriter();
var options = new CodeGeneratorOptions();
cp.GenerateCodeFromStatement(stmt, writer, options);
⇒writer.ToString() // "Contract.Requires((i > 1));"
```

T4

[1] [FCL, System.CodeDom Namespace] http://msdn.microsoft.com/en-us/library/system.codedom.aspx

Technologie T4 [T4] se používá ve Visual Studiu k vytváření šablon, které mohou sloužit jako preprocesor na vygenerování kódu tříd, které jsou pak zkompilovány. Druhá možnost použití šablon je na generování textu v programu - šablona (výpis 8.2) je převedena na třídu, jejíž metoda `TransformText` (výpis 8.3) spustí kód uvedený v šabloně a vygeneruje řetězec (výpis 8.4). Použití šablony je výhodné především tam, kde převažuje její textová část. Pro použití v této práci pro generování tříd by však musela být velmi jemně strukturovaná, čímž by ztratila na přehlednosti. Naprosto nevhodné by bylo šablony použít pro generování výrazů, které mohou být hluboce zanořené a přitom generují jen velice malé množství textu.

Výpis 8.2 Příklad šablony T4

```
<#@ template language="C#" #>
class <#= name #>
{
<# foreach (var property in properties) { #>
    public <#= property.Value #> <#= property.Key #> { get; set; }
<# } #>
}
```

Výpis 8.3 Příklad použití šablony T4

```
partial class ClassTemplate{
    private string name;
    private Dictionary<string, string> properties;
    public ClassTemplate(string name, Dictionary<string, string> properties){
        this.name = name; this.properties = properties;
    }
}
var properties = new Dictionary<string, string>();
properties["A"] = "int";
properties["B"] = "string";
var template = new ClassTemplate("Class1", properties);
⇒ template.TransformText()
```

Výpis 8.4 Příklad výstupu šablony T4

```
class Class1 {
    public int A { get; set; }
    public string B { get; set; }
}
```

Roslyn

Roslyn (.NET Compiler Platform, [ROS]) je překladač jazyka C# (a Visual Basic) implementovaný v jazyce C# (resp. Visual Basic). V dubnu 2014 byly společností Microsoft zdrojové kódy projektu uvolněny pod licencí Apache a projekt pokračuje ve vývoji jako open-source. Roslyn poskytuje API pro přístup k jednotlivým fázím překladu, což umožňuje nad ním implementovat služby pro vývojáře

jako formátování kódu, automatické doplňování nebo refaktoring. Celý projekt je zatím ve fázi Preview.

Pomocí Syntax Tree API lze sestavit syntaktický strom jazyka C# a pak vygenerovat odpovídající kód. Příklad ve výpisu 8.5 ukazuje generování příkazu `Contract.Requires(i>1);` pomoci statických metod třídy `SyntaxFactory`. API je velmi nízkoúrovňové, je potřeba řešit uzávorkování výrazů a kód na závěr zformátovat, jinak by neobsahoval bílé znaky.

Výpis 8.5 Příklad generování kódu pomocí `SyntaxFactory` z projektu Roslyn

```
// Identifikátory
var contract = SyntaxFactory.IdentifierName(SyntaxFactory.Identifier("Contract"));
var requires = SyntaxFactory.IdentifierName(SyntaxFactory.Identifier("Requires"));
var i = SyntaxFactory.IdentifierName(SyntaxFactory.Identifier("i"));
// Metoda Contract.Requires
var contractRequires = SyntaxFactory.MemberAccessExpression(
    SyntaxKind.SimpleMemberAccessExpression, contract, requires);
// Výraz i > 1
var one = SyntaxFactory.LiteralExpression(
    SyntaxKind.NumericLiteralExpression, SyntaxFactory.Literal(1));
var expr = SyntaxFactory.BinaryExpression(
    SyntaxKind.GreaterThanExpression, i, one);
// Volání metody
var arg_array = new ArgumentSyntax[]{SyntaxFactory.Argument(expr)};
var arg_list = SyntaxFactory.ArgumentList(SyntaxFactory.SeparatedList(arg_array));
var call = SyntaxFactory.InvocationExpression(contractRequires, arg_list);
// Příkaz
var statement = SyntaxFactory.ExpressionStatement(call);
// Formátování kódu
var formattedRoot = Formatter.Format(statement, new CustomWorkspace());
⇒ formattedRoot.ToFullString() // "Contract.Requires(i > 1);"
```

Vlastní implementace

Vzhledem k tomu, že žádný z uvedených způsobů není dostatečný z hlediska dostupné funkčnosti a přiměřené jednoduchosti použití, byla pro generování vytvořena jednoduchá implementace syntaxe (výpis 8.6) založená na třídě `StringBuilder`. Nachází se ve jmenném prostoru `Exolutio.CodeContracts.Code`. Podporuje všechny potřebné výrazy včetně lambda výrazů a automatické uzávorkování dle priority operátorů. Generování tříd, vlastností a metod je omezeno pouze na potřeby této práce.

Výpis 8.6 Příklad použitého generování kódu

```
// Metoda Contract.Requires
var contract = new Code.Identifier("Contract");
var requires = new Code.Identifier("Requires");
var contractRequires = new Code.MemberAccess(contract, requires);
// Výraz i > 1
var one = new Code.Int32Literal(1);
var i = new Code.Identifier("i");
var expr = new Code.BinaryOperator(
    BinaryOperator.OperatorKind.Greater, i, one);
// Volání metody
var call = new Code.Call(contractRequires, expr);
⇒ call.ToString(); // "Contract.Requires(i > 1);"
```

8.3 Sestavení programu

Program sestavíme ve vývojovém prostředí Visual Studio 2012 (nebo novějším) pro Microsoft .NET Framework 4.5. Otevřeme soubor `src\eXolutio.sln` z přílohy A a sestavíme běžným způsobem. Po případných změnách souborů s gramatikou jazyka OCL (`src\Model\OCL\Grammar\Syntax.g3` a `src\Model\OCL\Grammar\AST.g3`) je nutné přegenerovat zdrojové kódy parseru spuštěním dávkového souboru `src\ExternalBinaries\Antlr\generate.bat`.

8.4 Knihovna OclRuntime

Implementace typů OCL a podpůrné metody jsou umístěny do knihovny OclRuntime, která musí být vložena do vygenerovaného programu (`bin\OclRuntime.dll` v příloze A). V implementaci jsou použity kolekce z knihovny `System.Collections.Immutable`, takže ta také musí být přidána.

Všechny třídy se nacházejí ve jmenném prostoru `Exolutio.OclRuntime` a jsou to:

- implementace jednotlivých typů standardní knihovny (třída `OclAny` a odvozené),
- reprezentace typů jazyka OCL a jejich kompatibility (třída `OclClassifier` a odvozené),
- statická třída `Ocl` poskytující statické metody používané v generovaných výrazech,
- výjimky které mohou nastat při vyhodnocování výrazů (třída `OclException` a odvozené).

8.5 Testy pro program NUnit

Zdrojové kódy programu eXolutio obsahují podprojekt `Tests`, ve kterém jsou implementovány testy různých částí programu pro nástroj NUnit[2]. Tento projekt byl rozšířen o nové jednotkové testy knihovny OclRuntime (`Exolutio.Tests.OclRuntime`) a testy generování kódu, překladu samostatných výrazů jazyka OCL a překladu modelu a integritních omezení do jazyka C# (`Exolutio.Tests.CodeContracts`). Testy spustíme v programu NUnit po otevření projektu `src\Tests\Testsx.nunit` z přílohy A.

8.6 Dokumentace

Zdrojové kódy projektu knihovny OclRuntime obsahují dokumentační komentáře, z nichž je programem Sandcastle Help File Builder[3] vygenerována programátorská dokumentace rozhraní (`doc\OclRutime.chm` v příloze A). V `doc\CodeContracts.chm` je dokumentace rozhraní pro překlad.

8.7 Ukázkové projekty

Projekt pro program eXolutio obsahující model z obrázku 2.1 a další ukázkové projekty lze nalézt v `src\Projects\CodeContracts` v příloze A.

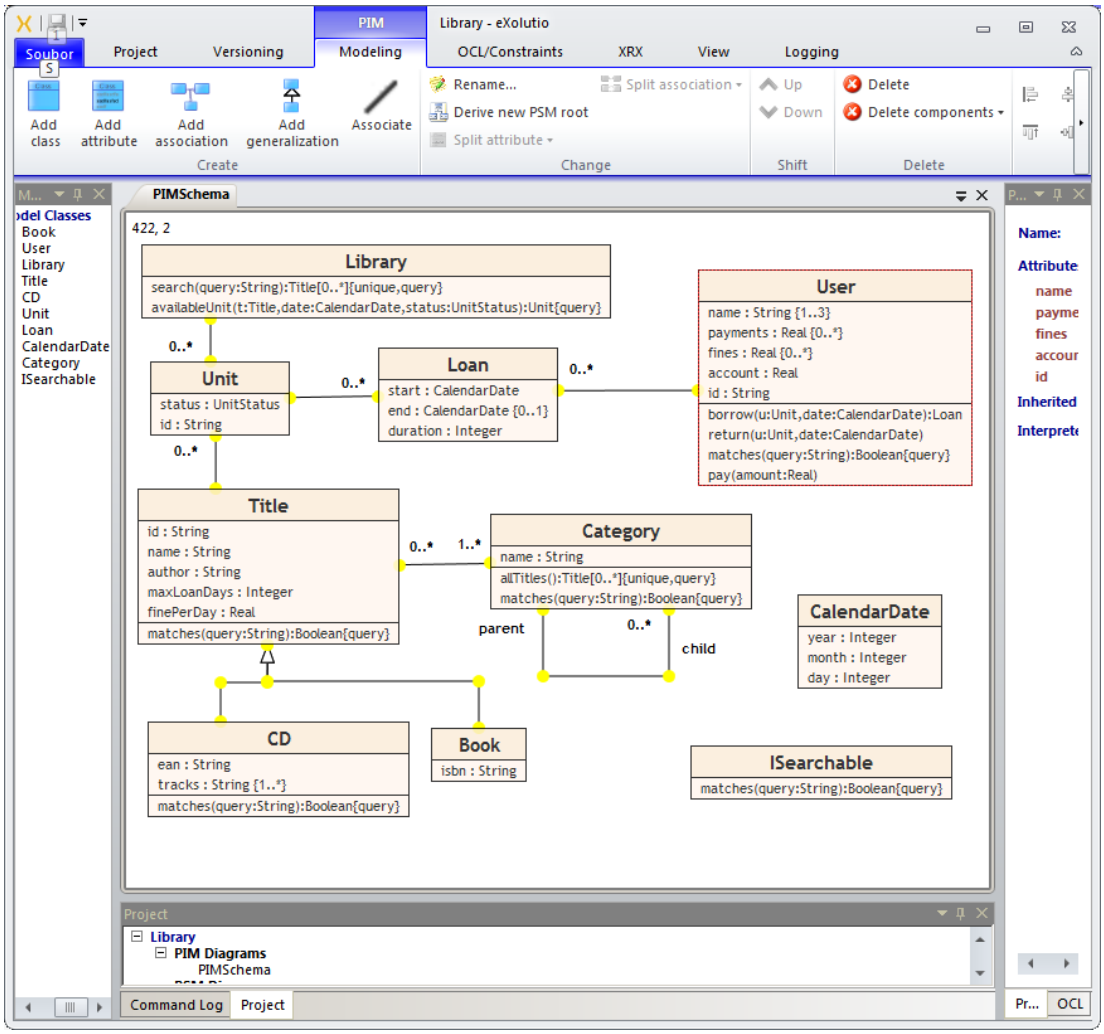
[2] http://www.nunit.org/
[3] https://shfb.codeplex.com/

9. Uživatelská dokumentace

9.1 Spuštění programu

Program ke svému běhu vyžaduje operační systém Windows a Microsoft .NET Framework 4.5. Program spustíme ze souboru `bin\Exolutio.WPFClient.exe` (příloha A).

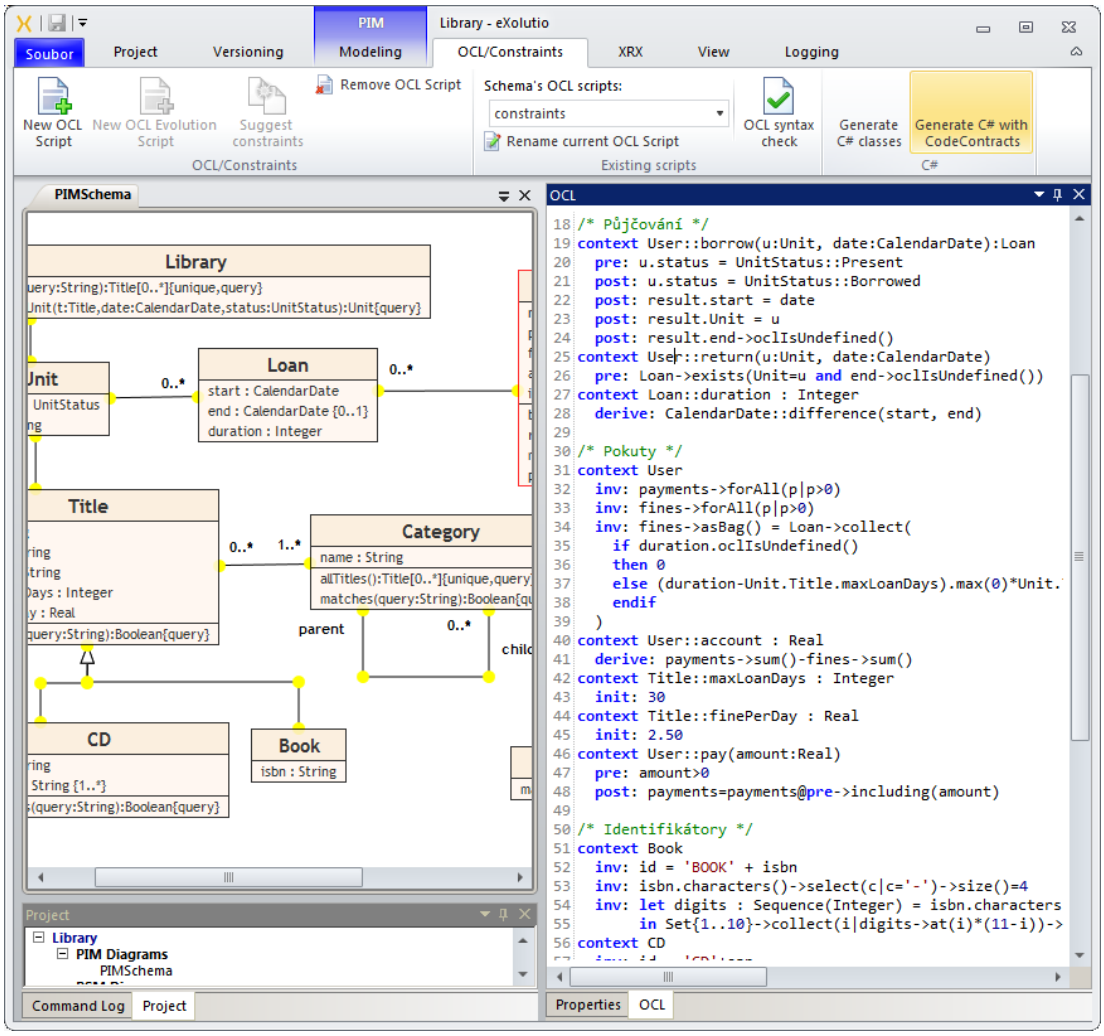
9.2 Použití



Obrázek 9.1: Diagram tříd v programu eXolutio

Uprostřed hlavního okna programu (obrázek 9.1) je grafický editor diagramu tříd modelu PIM. Pomocí panelu nástrojů (záložka „PIM“ / „Modeling“) a kontextových nabídek je možné do modelu přidávat nové třídy, atributy a asociace. Pro potřeby práce byl model rozšířen o nové elementy, které se v editoru nezobra-

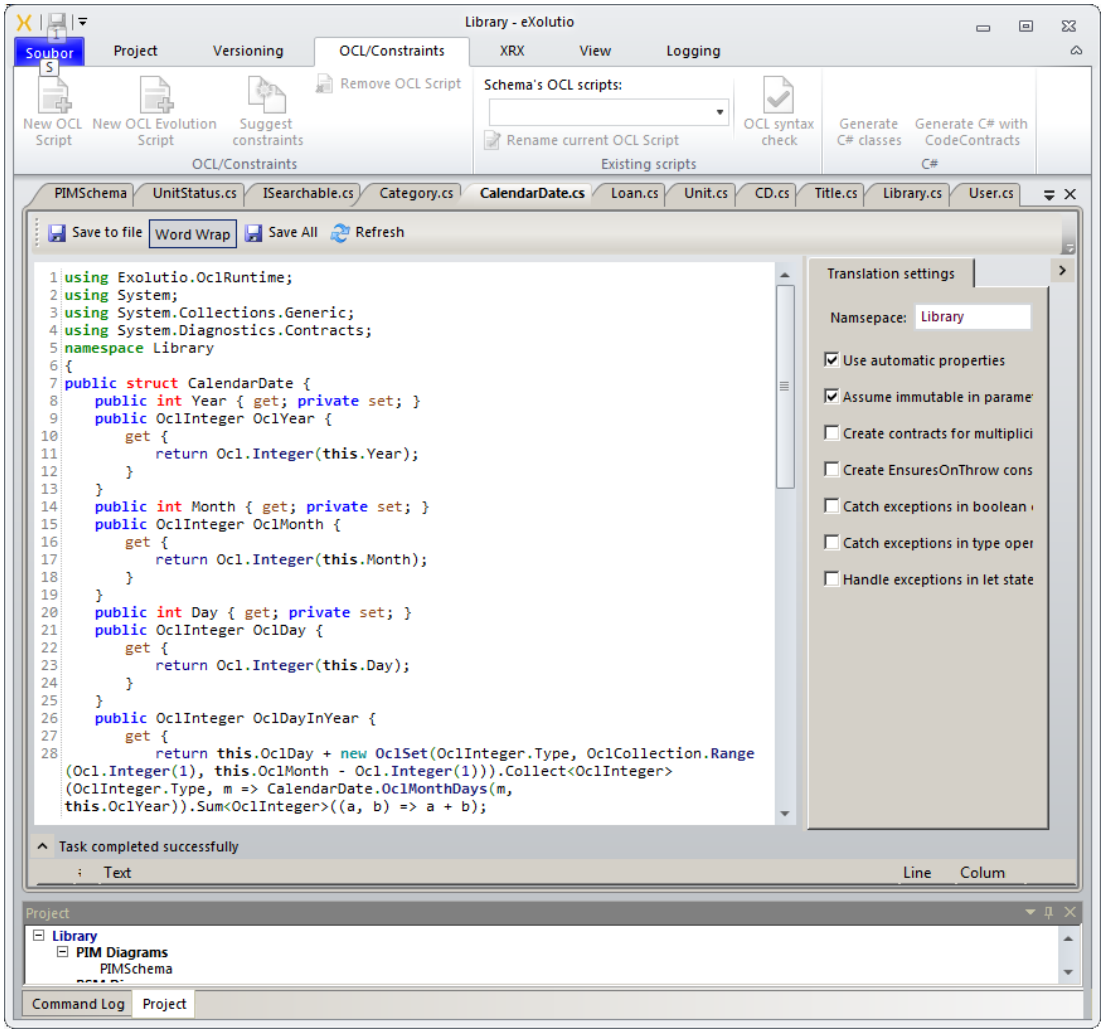
zují. Operace definované na třídách se v digramu zobrazují, ale nelze je vytvářet ani upravovat[1].



Obrázek 9.2: Skript jazyka OCL v programu eXolutio

Na záložce „OCL/Constraints“ můžeme vytvářet OCL skripty přidružené k modelu. Skript vybraný v rozbalovacím seznamu je zobrazen v editoru v pravé části okna (obrázek 9.2). Tlačítko „Generate C# classes“ vygeneruje z modelu zdrojové kódy v jazyce C# bez uvažování OCL, tedy bez Code Contracts. Tlačítko „Generate C# with CodeContracts“ vygeneruje zdrojové kódy včetně integritních omezení ze všech OCL skriptů modelu.

[1]Tyto elementy byly do ukázkových projektů přidány editací souboru `.exo` v editoru XML.



Obrázek 9.3: Vygenerovaný kód v programu eXolutio

Po vygenerování kódu se v prostřední části (obrázek 9.3) otevřou karty s jednotlivými vygenerovanými soubory (1 soubor pro každou třídu). V dolní části je seznam chyb, které nastaly při parsování OCL skriptů nebo při překladu. V každé kartě vpravo lze rozbalit panel s volbami překladu (popsány v sekci 9.2.1). Volby jsou společné pro všechny soubory a po změně se všechny vygenerované zdrojové kódy aktualizují. V případě úprav diagramu nebo skriptu je nutné nechat zdrojové kódy znovu vygenerovat tlačítkem „Referesh“. Tlačítky „Save to file“ resp. „Save All“ je pak možné uložit jednotlivé soubory, resp. všechny vygenerované soubory.

9.2.1 Volby překladu

Namespace Jmenný prostor pro vygenerované typy.

Use automatic properties Povolit generování automaticky implementovaných vlastností.

Assume immutable in parameters Předpokládat, že vstupní parametry metod se nemění a není třeba k nim ve výstupních podmínkách přistupovat přes volání metody `Contract.OldValue`.

Create contracts for multiplicities Vytvořit kontrakty pro kardinalitu kolekcí ve vlastnostech, parametrech a návratových hodnotách.

Create EnsuresOnThrow constraint Pro výstupní podmínky vytvořit i variantu pro případ, že je metoda opuštěna vyhozením výjimky.

Catch exceptions in boolean operators Korektní zpracování hodnoty `invalid` (zachycení výjimek) v operacích typu `Boolean`.

Catch exceptions in type operations Korektní zpracování hodnoty `invalid` (zachycení výjimek) v operaci `oclType`.

Handle exceptions in let statements Korektní zpracování hodnoty `invalid` v konstrukci `let`.

10. Závěr

Ze srovnání jazyka OCL a technologie Code Contracts v kapitole 6 je patrné, že oba umožňují specifikaci jako vstupních a výstupních podmínky operací a invariantů tříd. OCL navíc obsahuje konstrukce (`body`, `init`, `derive`, `def`), které lze reprezentovat ve vygenerovaném kódu v jazyce C#, ovšem bez využití Code Contracts. I tyto byly v rámci práce implementovány.

Naopak Code Contracts umožňují podrobné nastavení způsobu ověřování akcí provedených při porušení integritních omezení, kdežto OCL tuto problematiku neřeší. V Code Contracts je také potřeba obcházet některá omezení vyplývající z návrhu jazyka C# a platformy .NET (např. nutnost použití `ContractClass` pro některé typy).

Standardní knihovny jazyka OCL obsahuje primitivní typy, které mají obdobu v knihovně BCL, jejich sémantika je v některých případech definována značně odlišně. Pro implementaci kolekcí lze v C# využít rozsáhlých možností rozšiřujících metod LINQ, vhodné je také použití neměnných kolekcí `System.Collections.Immutable`. Odlišnosti typových systémů OCL a C# výrazně komplikují implementaci v jazyce C#. Překonání těchto problémů se podařilo díky vhodnému využití generických metod a lambda výrazů.

Snaha o dodržení sémantiky zdrojového jazyka, v představené implementaci vyřešená konverzí všech hodnot ve výrazu na typy z knihovny OclRuntime, způsobila, že integritní omezení jsou přeložena na poměrně komplikované výrazy, které nejsou vhodné pro statickou analýzu. Na druhou stranu implementace všech operací standardní knihovny jazyka OCL v knihovně OclRuntime umožnila

Chyby ve specifikaci jazyka OCL představují problém pro porozumění specifikaci a implementaci nástrojů. V průběhu práce vyšla nová verze specifikace, která velké množství chyb opravila, proto jsme na to zareagovali implementací této nové verze. Některé chyby ale stále zůstávají otevřené [RTF], nově jsme nahlásili chybu v definici iterátoru `sortedBy`.

Diagramy tříd a parser jazyka OCL v programu eXolutio musely být doplněny o podporu některých konstrukcí (parsování integritních omezení pro operace, `def`, rozhraní a výčtové typy, statické operace a vlastnosti), je zde ovšem velký prostor pro vylepšení, například v uživatelském rozhraní, které aktuálně neumožňuje provádět na modelu všechny potřebné úpravy (například přidávat a upravovat operace).

10.1 Budoucnost

10.1.1 Možnosti dalšího vývoje

Pro zvýšení potenciálu praktického využití práce by bylo vhodné především:

- doplnění editoru diagramu tříd o nové elementy, aby kompletní model mohl být vytvořen v grafickém uživatelském rozhraní,
- doplnění modelu o další elementy jazyka UML, aby bylo možné využít všech možností cílového jazyka C#,
- redukce složitosti generovaných výrazů:
  - nalezení jednodušších ekvivalentů pro některé často používané konstrukce,
  - nalezení podmínek, za kterých lze použít pro některé konstrukce jednodušší implementaci (např. propagace konstant, vynechání ošetření hodnot `null` a `invalid` v místech, kde se nemohou vyskytnout),
  - přizpůsobení generovaných výrazů možnostem statického analyzátoru technologie Code Contracts.

10.1.2 Jazyky OCL a C#

Autoři jazyků OCL a C# stále pracují na jejich vylepšování a snaží se je přizpůsobit potřebám jejich uživatelů. Dosud zveřejněné návrhy obsahují i nové vlastnosti, které by mohly usnadnit překlad mezi těmito jazyky.

C# 6.0

Díky projektu Roslyn je již možné vyzkoušet některé nové konstrukce, které se mohou objevit ve verzi C# 6.0, na internetových stránkách projektu najdeme tabulku všech plánovaných vlastností [CS6].

- Syntaxe primárního konstruktoru a inicializace automatických proměnných zjednoduší tvorbu neměnných (immutable) typů a redukuje nutnost psaní často se opakujícího podpůrného kódu.
- Výraz jako tělo metody nebo vlastnosti poskytuje přehlednější syntaxi pro operace a vlastnosti definované pomocí konstrukcí `der`, `body` nebo `def`.
- Deklarace proměnných ve výrazech umožní jednodušší volání metod s výstupními parametry a vyjádření konstrukce `let` ve výrazech.

- Operátor postupného vyhodnocení `;`, jenž bude obdobou operátoru `,` z jazyka C, umožní mimo jiné uvnitř výrazu volat metody s návratovým typem `void`. To by umožnilo volání operací vracejících `OclVoid` bez použití obalovací metody.

Výpis 10.1 Ukázka využití potenciálních vlastností jazyka C# 6.0

```
public struct CalendarDate (int year, int month, int day){
  public int Year { get; } = year;
  public int Month { get; } = month;
  public int Day { get; } = day;
  ...
  [Pure]
  public static bool IsLeapYear(int y) => y%4==0 && (y%100!=0 || y%400==0);
}
```

OCL 2.5

V současné době probíhají přípravy verze 2.5 jazyka OCL. Výzva k zasílání návrhů[RFP, kapitola 6] popisuje témata, kterým by se návrhy měly věnovat, především současné problémy jazyka OCL.

Ve zprávě [AA] jsou naznačeny další možnosti vývoje jazyka OCL. Zmíněna je například podpora generických typů, konstrukce objektů, úpravy sémantiky přetypování, změna vztahu typu `OclAny` a kolekcí, uvažování reprezentace primitivních typů, nebo specifikace gramatiky ve strojově čitelném formátu.

Literatura

[AA] *Report on the Aachen OCL Meeting* [online]. [cit. 9. 7. 2014]. Dostupné z: http://www.db.informatik.uni-bremen.de/publications/intern/aachen-2013-nov/paper.pdf.

[CC] *Code Contracts User Manual* [online]. Microsoft Corporation. [cit. 9. 7. 2014]. Dostupné z: http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf.

[CCW] *Code Contracts* [online]. Microsoft Corporation. [cit. 27. 7. 2014]. Dostupné z: http://research.microsoft.com/en-us/projects/contracts/.

[CS] *C# Language Specification Version 5.0* [online]. [cit. 9. 7. 2014]. Dostupné z: http://www.microsoft.com/en-us/download/confirmation.aspx?id=7029.

[CS6] *.NET Compiler Platform ("Roslyn") Language feature implementation status* [online]. [cit. 12. 7. 2014]. Dostupné z: https://roslyn.codeplex.com/wikipage?title=Language%20Feature%20Status&referringTitle=Documentation.

[DOC] *Dresden OCL* [online]. [cit. 27. 7. 2014]. Dostupné z: http://www.dresden-ocl.org/index.php/DresdenOCL.

[EUI] *OCL Documentation, Users Guide, User Interface* [online]. [cit. 27. 7. 2014]. Dostupné z: http://help.eclipse.org/juno/topic/org.eclipse.ocl.doc/help/UserInterface.html#CodeGenerationMode.

[FCL] *.NET Framework Class Library* [online]. [cit. 9. 7. 2014]. Dostupné z: http://msdn.microsoft.com/en-us/library/gg145045%28v=vs.110%29.aspx.

[MDT] *Model Development Tools (MDT)* [online]. The Eclipse Foundation. [cit. 27. 7. 2014]. Dostupné z: http://www.eclipse.org/modeling/mdt/?project=ocl.

[OCL] *Object Constraint Language Version 2.4* [online]. Object Management Group. [cit. 9. 7. 2014]. Dostupné z: http://www.omg.org/cgi-bin/doc?formal/2014-02-03.pdf.

[RFP] *Object Constraint Language (OCL) 2.5 Request For Proposal* [online]. Object Management Group. [cit. 9. 7. 2014]. Dostupné z: http://www.omg.org/cgi-bin/doc?ad/2014-03-05.

[ROS] *.NET Compiler Platform ("Roslyn")* [online]. Microsoft Corporation. [cit. 27. 7. 2014]. Dostupné z: http://roslyn.codeplex.com/.

[RTF] *Issues for OCL 2.4 Revision Task Force mailing list* [online]. [cit. 9. 7. 2014]. Dostupné z: http://www.omg.org/issues/ocl2-rtf.html.

[T4] *Code Generation and T4 Text Templates* [online]. Microsoft Corporation. [cit. 27. 7. 2014]. Dostupné z: http://msdn.microsoft.com/en-us/library/bb126445%28v=vs.110%29.aspx.

[UML] *OMG Unified Modeling Language (OMG UML), Superstructure Version 2.4.1* [online]. Object Management Group. [cit. 9. 7. 2014]. Dostupné z: http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.

[XRG] *XML Research Group | XRG - Software* [online]. [cit. 27. 7. 2014]. Dostupné z: http://xrg.ksi.ms.mff.cuni.cz/software/.

[Arnold] Arnold, D. *C# / OCL Compiler - Version 1.01 (April 25th 2004)* [online]. [cit. 27. 7. 2014]. Dostupné z: http://www.davearnold.ca/csocl/.

[Lippert] Lippert, E. *Mutating readonly structs* [online]. [cit. 27. 7. 2014]. Dostupné z: http://ericlippert.com/2008/05/14/mutating-readonly-structs/.

A. Přílohy

Obsah přiloženého CD:

- `text.pdf` Text práce
- `src\` Zdrojové kódy
  - `CodeContracts\` Implementace překladu
  - `OclRuntime\` Běhová knihovna
  - `Tests\CodeContracts\` Testy překladu
  - `Tests\OclRuntime\` Testy běhové knihovny
  - `Projects\CodeContracts\` Ukázkové projekty
- `bin\` Binární soubory
  - `Exolutio.WPFClient.exe` Hlavní program
- `doc\` Generovaná dokumentace
  - `OclRuntime.chm` Dokumentace knihovny OclRuntime
  - `CodeContracts.chm` Dokumentace API překladu

Seznam zkratek

OMG Object Management Group

LINQ Language Integrated Query

OCL Object Constraint Language

UML Unified Modeling Language

XRG XML Research Group

XML Extensible Markup Language

SQL Structured Query Language

PIM Platform independent model

BCL Base Class Library

NaN Not a number

WPF Windows Presentation Framework

ISBN International Standard Book Number