PROGRAMMING LANGUAGE C.C

Vojtěch Merunka
Czech University of Life Sciences Prague, merunka@pef.czu.cz

ABSTRACT

The C.C programming language is outcome of our research and has been implemented in the system analysis and modeling tool Craft.CASE of a British software company. This language is designed for automated modeling such as simulations, MDA support, domain-specific capabilities, flexibility, consistency and integrity checking, reporting etc. Interesting side effect of this programming language is its feasible application as a first teaching language in algorithmization and programming courses. Author regrets to announce this paper is published in English. It is caused by the National research and development policy issued by the Research and Development Council of the Czech Republic, that discriminates the Czech language.

KEYWORDS

scripting languages, C.C language, Craft.CASE tool, design patterns, re-factoring, class normalization, UML

INTRODUCTION

In this paper, we introduce the C.C language, show its basic concepts, syntax and demonstrate the way in which it cooperates with the Craft.CASE modeling tool.

The C.C language design is an outcome of our research. Interpreter of this language has been recently included into the Craft.CASE modeling tool developed by the British Company CRAFT.CASE Ltd. [2] This company thus takes all activities which were connected with the Craft.CASE and the BORM method in the past, including their future advancements.

SCRIPTING AND AUTOMATED MODELLING

There are several ways how to classify these techniques. For example, Jean-Marc Jezequel [7,8] presents the following classification:

- General purpose programming languages - Java, VB, C++, C#, etc. Rules and scheduling are implemented from scratch using the programming language.
- Generic transformation tools - XSLT transformation and graph transformation tools.
- CASE tools scripting languages - for example Arcstyler, Objecteering, OptimalJ, or Fujaba.
- Dedicated model transformation tools - for example OMG QVT style.
- Meta Metamodeling tools - for example MetaEdit+, XMF-Mosaic, or KerMeta.

The Craft.CASE modeling tool provides model transformation via the C.C interpreter. Our approach respects categories 1 and 3, but is more universal, because it enables not only scripting and rule implementation. In addition, the C.C interpreter is able to perform all operations on model (including simulations, refactoring, new diagram creation, etc.), that are executed manually by users from graphical user interface. On the other hand, the language is not standardized on the present, thereby it is not possible to share the source code with other modeling tools.

The C.C language is a functional programming language with PASCAL-like syntax with several imperative constructs and some features coming from languages PROLOG, Erlang, Ruby, Python and Smalltalk. It has an interpreted programming environment. C.C is used for following purposes:

- As a scripting language. Procedures in C.C are able to pass through project database and compose miscellaneous documentation reports.
- Precise process simulation. Procedures in C.C can compute various simulation data, control simulation flow, etc.
- Automated manipulations with model (e.g. applying design patterns, refactoring and class normalization).
- Consistency and integrity check of project database. This feature covers the same functionality as the OCL.
- Data export in different formats (namely XMI and binary formats of other CASE tools).
- Data import from different data sources (e.g. ODBC, CSV etc.).

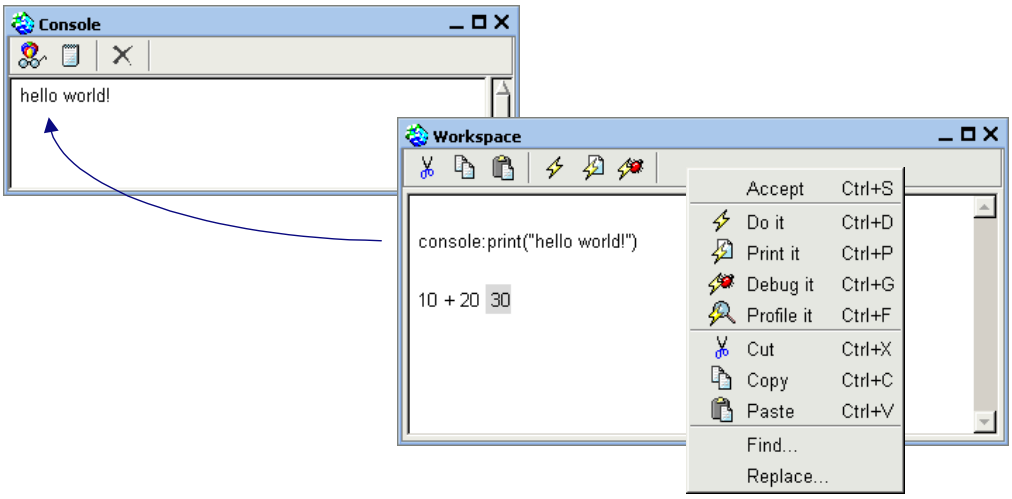
LANGUAGE ARCHITECTURE

The C.C interpreter consists of *workspace* (expression evaluator), *console*, module *browser*, time *profiler* and *debugger* (for tuning and tracing). The language has following features:

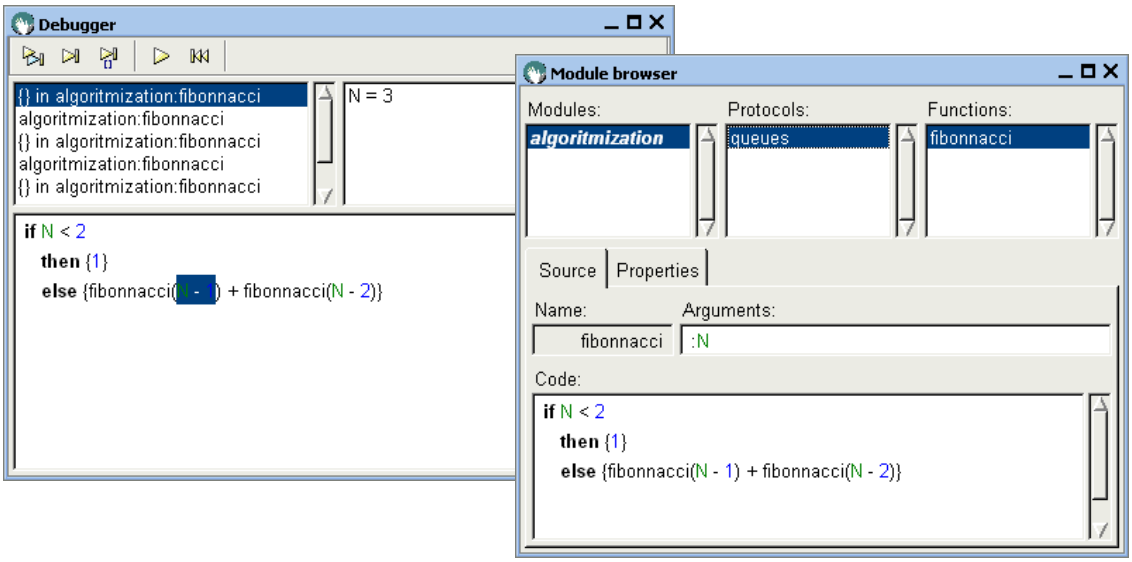
- The C.C architecture consists of *modules* having *functions*. Modules are both system built-in and user-written.
- Variables must begin with capital letters.
- Built-in values are true, false, nil, e (Euler's number), i (purely imaginary number), pi (Ludolf's number), infinity, tiny (infinitesimal zero) and a lot of functions in miscellaneous modules.
- The only types are:
  1) Symbol (atomic textual values beginning with non-capital letters).
  2) String of characters written in double quotation marks.
  3) Number (incl. complex numbers).
  4) Date.
  5) Time.
  6) Logical value as predefined symbols true and false.
  7) nil.
  8) Collection of elements of any type. There are three types of collections: list, set and dictionary. Elements of collections are accessible through box brackets.
  9) Function as lambda-expression [5] that is written in curly brackets. To illustrate, lambda-expression ($\lambda x\, \lambda x \mid x^2 + y$) is written as {:X , :Y | X ^ 2 + Y}.

Following line of the C.C code implements a hello world program.

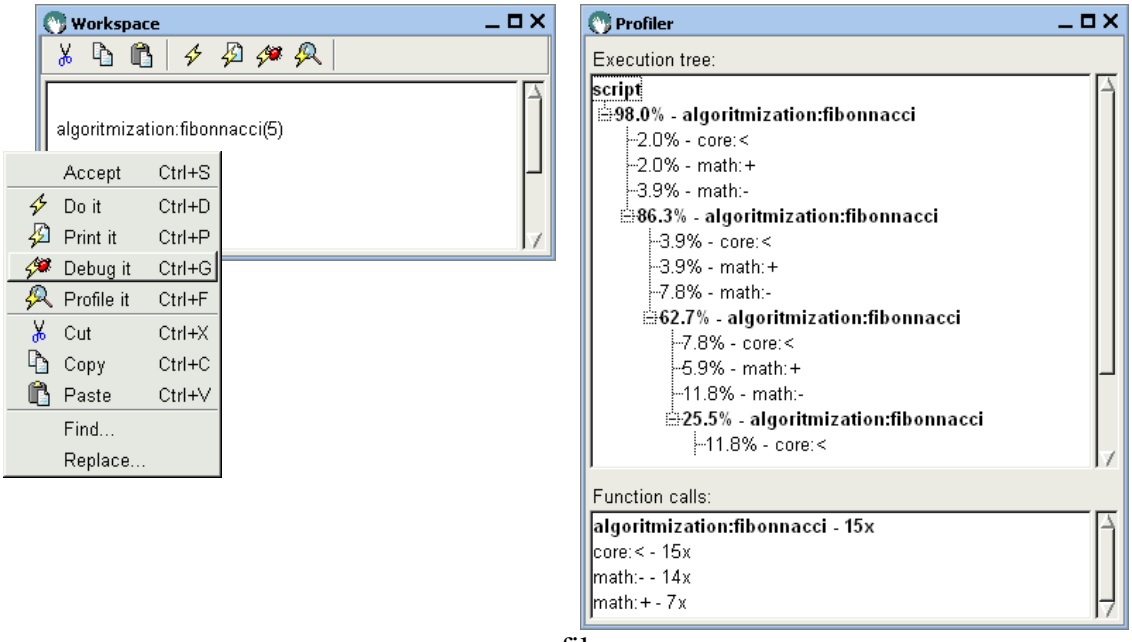
```
console:print("Hello world!").
```



console and workspace



debugger and module browser



profiler

Functions
Each C.C function must be a member of some module. Therefore previous example of a hello world program working with module named console and function named print can be written as follows.

```
| M , F |                 # declaration of variable names
M := console.             # assigning symbol "console" to var "M"
F := print.               # assigning symbol "print" to var "F"
M:F("hello world!").      # function call
```

User-defined functions are represented by function expressions stored in variables. For example function $F(x, y) = 10x + y$ can be implemented as

```
F := {:X , :Y | 10*X + Y}.
```

This function F can be applied on arguments via round brackets as for example F(3,4). However, this function call can be used directly without the need to store it in any variable, like

```
{:X , :Y | 10*X + Y}(3,4).
```

Additionally, there are yet some advanced features related to default values of lambda-variables, order of parameters in function call and possibility to call function with incomplete set of parameters.

Collections
Following example shows declaration of a list L and a dictionary D.

```
L := [10 , 20 , 30 , 40 , 50].      D := [first := 10 , second := 20].
```

Then we can access elements of these collections as follows.

```
L[1] = 10.      L[2] = 20.      D[first] = 10.      D[second] = 20.
```

We have have defined nine operators for comfortable collection processing. Nonetheless these operators (and all other C.C operators) are interpreted as functions as well.

- adding: *collection* add *element*, *collection* add-all *collection*.
- removing: *collection* remove *element*, *collection* remove-all *collection*.
- set operations: *collection* intersection *collection*, *collection* union *collection*.
- testing: *value* in *collection*.
- selection: *collection* // *function*.
- projection: *collection* >> *function*.

Selection and projection is explained in this example:

```
[10,20,30,40,50] // {:X | X > 20}     =    [30,40,50].
[10,20,30,40,50] >> {:X | X + 1}     =    [11,21,31,41,51].
```

Craft.CASE model elements behaves as collections as well. For example, if there is an element AClass, then the expression AClass[prop-name] := NewValue changes a value of the property prop-name of this element.

Control structures

Control structures are realized by operators, but they have internally the same interpretation as functions. They are:

1) if *logical-expression* then *function* [else *function*] .
2) for *collection* do *function* .
3) from *value* to *value* [by *value*] do *function* .
4) repeat *function* until *function-with-logical-expression* .
5) while *function-with-logical-expression* do *function* .

Following two pieces of code shows the same iteration:

```
for [10,20,30,40,50] do {:X | console:print-nl(X)}.
```

```
| X |
X := 10.
while not(X > 50)  do {console:print-nl(X). X := X + 10}.
```

Code examples

The C.C language is a universal programming language. Here are two small examples of well-known algorithms.

```
##### recursive definition of factorial #####
| Factorial |
Factorial := {:X |  if  X = 0
                    then  {1}
                    else  {X * Factorial(X - 1)}}.
```

```
##### Eratosthenes' generator of prime numbers #####
| Max , Non-primes , Primes |
Max := integer(dialog:request("maximum number?")).
Non-primes := set:new().
Primes     := list:new().
from 2 to Max do {:N |  if not(N in Non-primes)
                        then {Primes add N.
                              from N
                              to Max
                              by N
                              do {:N1 | Non-primes add N1}}}.
console:print(Primes).
```

Path expression

Path expression (using operator „–>“) is an implementation of the graph traversing algorithm. It contrives to collect neighbors of an element or a collection of elements in the project database with respect to the Craft.CASE metamodel. This metamodel is based on graph concept consisting from *nodes* and *links*. Each *link* is one-way oriented and has one *source*

node and one *target node*. The whole Craft.CASE project is a *node* as well. If this project consists of diagrams, they are *nodes* linked to this project. If a diagram consist of elements, they are *nodes* linked to this diagram. Of course, miscellaneous relations between elements in particular diagrams are *links* between corresponding *nodes* too.

To illustrate, if there is a class Person, we can access all methods of all classes inheriting from this class as follows:

```
Person –> "subtype" –> "Class" –> "Method".
```

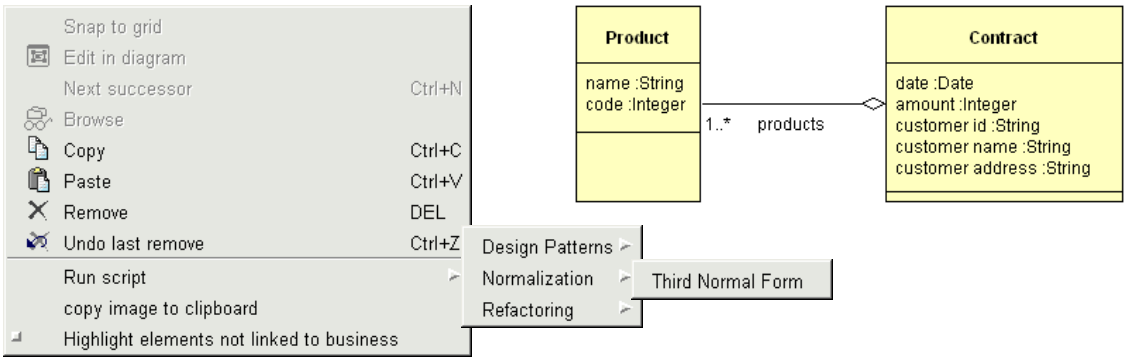
Modules
The C.C interpreter has several built-in modules concerning in mathematics, simulation, diagramming, data input and output, reporting etc. It is possible to declare, that the C.C interpreter offers the same behavior as the human interface of the Craft.CASE including new diagram creation, symbol editing etc.

MODELING EXAMPLES

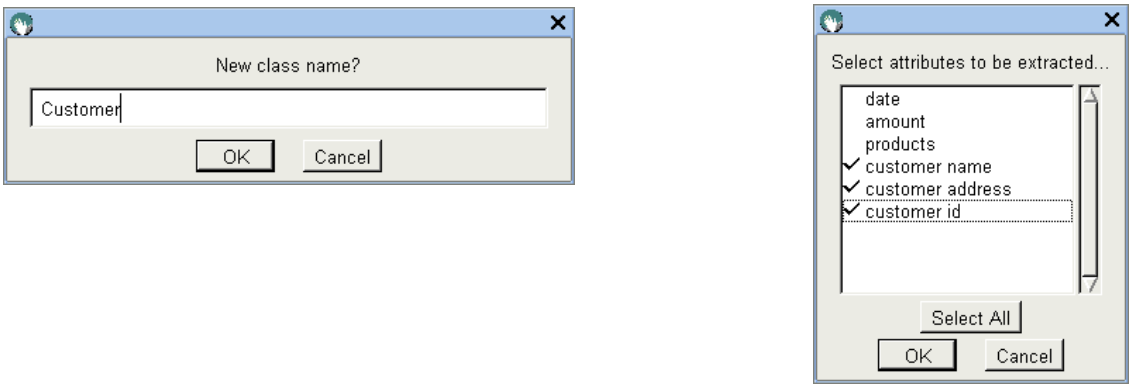
Our experience denotes the fact that the design pattern technique, the object normalization technique and refactoring technique share common principle of model transformation. Hence all these techniques can be automated through C.C code with a project database. In this chapter we demonstrate practical examples of this idea.

Refactoring
It is possible to define refactoring as any sequence of system transformations, where behavior of the system remains unchanged. (An exception might be for instance a slightly different delay between user impulse and subsequent system response, nevertheless from user point of view refactoring has actually no importance.) From system modeling aspect, refactoring is performed for optimalization, reusability and maintainability reasons [8]. Classical book on refactoring is [3].
In the following piece of code we present an interactive algorithm for interactive creating a new super-class to selected classes from a conceptual class-diagram.

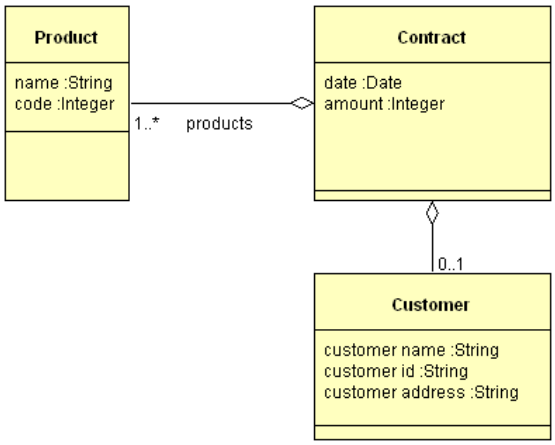
```
| Classes , NewClass |
# which are classes from selection?
Classes := editor:selection() // "Class".
if list:is-empty(Classes) then {return dialog:warn("No classes selected!")}.

# create new class, name it and add it into diagram
NewClass := project:new-node("Class").
NewClass[name] := dialog:request("New class name?").
editor:add-element(NewClass).

# assign new class as superclass of selected classes
for Classes do {:Class | editor:add-new-link(Class , NewClass , "Supertype")}.
```



Third normal form example – initial situation



Third normal form – interactive dialogues



Third normal form - result

Design patterns

Design patterns are proven solutions to design problems. A design pattern is a template for how to solve a problem. Design patterns are a technique for system design. Using of design pattern on some particular solution is time (and also money) saving. More detailed information about design patterns for software systems development and their classification is in book [4].

Currently all design patterns from this book are implemented as interactive functions in the C.C language. Moreover, the exploration of business process patterns and their subsequent implementation has been started. Following piece of code shows the implementation of *Adapter* pattern.

```
| Classes , Adaptee , Adapter , AdapterLink , Method |
# which are classes from selection?
Classes := editor:selection() // "Class".
if list:size(Classes) <> 1 then {return dialog:warn("Select one class to be adapted!")}.

# select adaptee class
Adaptee := Classes[1].

#create and link adapter class
Adapter := project:new-node("Class").
Adapter[name] := "Adapter".
editor:add-element(Adapter).

AdapterLink := project:new-link(Adapter, Adaptee , "Composition").
AdapterLink[name] := "adaptee".
AdapterLink[cardinality] := "1".
editor:add-element(AdapterLink).

Method := project:new-node("Method").
Method[name] := "Request()".
project:new-link(Adapter , Method,  "ownership (conceptual)").
```

Object normalization

Object normalization is similar approach to the relational data normalization. It is applied to object-oriented data model. There exist several approaches to object normalization. The most advanced information is in book by Scott Ambler [1], where object-oriented data modeling is discussed. In our technology, we use the Scott Ambler's three levels of object-oriented model normal forms. [9] (It is obvious, that "standard" relational normal forms are implementable by a C.C code as well.) Following piece of code shows our implementation of the third normal form.

```
| Classes , OldClass , NewClassName , NewClass , AttributeNames ,
  RemovedAttributes , Link |
# which are classes from selection?
Classes := editor:selection() // "Class".
if set:is-empty(Classes) then {return dialog:warn("No classes selected!")}.
OldClass := set:any(Classes).

# create new class, name it and add it into diagram
NewClassName := dialog:request("New class name?" , "New Class").
NewClass := project:new-node("Class").
NewClass["name"] := NewClassName.
editor:add-element(NewClass).
editor:add-new-link(OldClass , NewClass , "Composition").

# select instance variables to be extracted from an old class to a new class
AttributeNames := dialog:choose-multiple
     ("Select attributes to be extracted..." , OldClass -> "Composition" >> {:X | X[name]}).

#remove them from an old class and remember them
```

```
RemovedAttributes := dictionary:new().
for (OldClass –> "Composition")
do {:Composition |
if Composition[name] in AttributeNames
then {project:remove-element(Composition).
      RemovedAttributes[Composition[name]] := element:target(Composition)}}.

#add attributes into a new class
for dictionary:keys(RemovedAttributes)
do {:Name |
     Link := project:new-link(NewClass, RemovedAttributes[Name] , "Composition").
     Link[name] := Name.}
```

CONCLUSION

The C.C language and Craft.CASE are instruments we use to support our research in area of systems modeling. [6] This platform is able to solve many problems with interconnection of business models and software models, business process simulations, step-by-step model transformations, domain-specific capabilities, model checking an reporting etc. The author would like to acknowledge the support of the research grant project MSM6046070904 of the Czech Ministry of Education, Youth and Sports.

REFERENCES

1. Ambler S.: *Building Object Applications That Work, Your Step-By-Step Handbook for Developing Robust Systems Using Object Technology*, Cambridge University Press/SIGS Books, 1997, ISBN 0521648262
2. *Craft.CASE Home Page* www.craftcase.com.
3. Fowler M.: *Refactoring.* Addison-Wesley 1999. ISBN 0-201-48567-2.
4. Gamma E., Helm R., Johnson R., Vlissides J.M.: *Design Patterns: Elements of Reusable Object-Oriented Software*, ISBN 0-201-63361-2
5. Hankin C.: *Lambda Calculi - A Guide for Computer Scientists, Clarendon Press - Oxford 1994*, ISBN 0-19-853841-3
6. Liu L., Roussev B. et al.; *Management of the Object-Oriented Development Process - Part 15: BORM Methodology*, ISBN 1-59140-605-6
7. Muller P-A., Fleurey F., and Jézéquel J-M.: *Weaving executability into object-oriented meta-languages*. in S. Kent L. Briand, editor, Proceedings of MODELS/UML'2005, volume 3713 of LNCS, pages 264-278, Montego Bay, Jamaica, October 2005. Springer.
8. Sunye G., Pollet D., Le Traon Y., Jézéquel J-M.: *Refactoring UML Models*.
9. Vrany J., Struska Z., Merunka V.: *Object normalization as the contribution to the area of formal methods of object-oriented database design*, in proceedings of the eighth International Conference on Enterprise Information Systems: Databases and Information Systems Integration ICEIS 2006, Paphos, Cyprus, INSTICC Press, 2006, vol. 3, p. 471-474. ISBN 972-8865-41-4.