The syntax of the OUN language

Olaf Owe
Department of Informatics, University of Oslo, Norway

February 21, 2002

Contents

1 The OUN language

This document defines the grammar of the *OUN* language ("Oslo University Notation"), using extended BNF. The language is designed for high-level programming and modeling of open distributed systems, with support of behavioral specification and modular program reasoning, and was developed in the context of the Adapt-FT project [1]. The language has similarities to the one suggested in [3] and is explained in more detail in [5]. Distributed units are represented by objects, each with its own (virtual) processor. Each object can handle a number of processes, corresponding to remaining parts of method activations. A method activation may be temporarily suspended by use of guards, with the syntax $guard \rightarrow statement$, allowing other enabled processes to continue. A suspended process is enabled when its initial guard is satisfied. A method call $m(in; out)$ can have several *in*-parameters as well as *out*-parameters. A remote call $m(in;out)$ **do** s **od** will execute the statments s while waiting for the callee o to complete the call (if different from this object).

Openness is supported by run-time class upgrades, called **class extension**, allowing a class to be changed at run-time without interrupting the execution. A class extension may modify an existing class by adding new fields and new methods, by redefining existing methods (as well as associated behavioral specification), and by providing support of additional interfaces.

Specifications of classes and interfaces are given by invariants and pre- and post-conditions, using predicates that may refer to the *communication history* **H**, following [3, 6, 4]. In additon to class and interfaces specification, we allow contract specifications for subsystems invloving several objects.

The language is object-oriented supporting single and multiple inheritance of classes as well as interfaces, late binding, overloading, dynamic generation of objects, and encapsulation. Remote access to fields are not allowed and inter-object communication is done by means of method interaction controlled by interfaces. Method interaction is represented by two-way asynchronous communication, letting each such communication event correspond to an event in the communication history. We use arrows to visualize the direction of the communication event.

Objects are representing concurrent units in a distributed setting, while local data structure is defined by data types, using a syntax similar to [2, 4]. The language is strongly typed, using co-interfaces (specified by **with** clauses) in order to be able to write type-correct call-backs. We refer to [5] for further details of the language.

Notational conventions

Terminal symbols appear in bold, square brackets are used for optional parts; {*something*} means that *something* may be repeated n times, $n \in \mathbb{N}$. We enclose terminal symbols in quotations marks when necessary to avoid confusion.

1.1 Interface and contract definition

spec_interface ::=
  interface *interface_name* ['['*type_bindings*']']['**(**'*object_bindings*'**)**']
    [**inherits** *interfaces*]
  begin
    [**types** *type_decls*]
    [**with** *interface_name*
      {*operation*}]
    [**asm** *expr*]
    [**inv** *expr*]
    [*auxiliary_part*]
  end

spec_contract ::=
  contract *contract_name*
  begin
    with *object_bindings*
    inv *expr*
    [*auxiliary_part*]

end

1.2 Class and subclass definition

spec_class ::=
  class *class_name* ['['*type_bindings*']']['**(**'*bindings*'**)**']
    [**implements** *inst_interfaces*]
    [**inherits** *classes*]
  begin
    [**types** *type_decls*]
    [**val** *constant_declarations*]
    [**var** *var_declarations*]
    [**init** *imperative_code*]
      {*operation* == *imperative_code*}
    {**with** *interface_name*
      {*operation* == *imperative_code*}
      [**asm** *expr*]}
    [**inv** *expr*]
    [*auxiliary_part*]
  end

1.3 Dynamic class extension

spec_class_extension ::=
  class extension *class_name*
    [**implements** *inst_interfaces*]
  begin
      {*operation* == *imperative_code*}
    {**with** *object_binding*
      {*operation* == *imperative_code*}
      [**asm** *expr*]}
    [**inv** *expr*]
    [*auxiliary_part*]
  end

auxiliary_part ::=
  func *function_sigs*
  [**def** *function_defs*]
  [**axiom** *exprs*]
  [**lma** *exprs*]

1.4 Basic elements: Lists and bindings

type_bindings	::=	{type_binding,} type_binding
object_bindings	::=	{object_binding,} object_binding
bindings	::=	{binding,} binding
type_binding	::=	type_name **:** type_deter
object_binding	::=	object_name **:** interface_name
binding	::=	identifier **:** type_name
var_declarations	::=	{var_declaration,} var_declaration
var_declaration	::=	binding [**:=** expr]
constant_declarations	::=	{var_declaration,} var_declaration
type_deter	::=	**Data_Type** | **Interface**
operation	::=	**opr** operation_name'('[in_out_parameters]')'
in_out_parameters	::=	[**in**] bindings [; **out** bindings] | **out** bindings
function_sigs	::=	{function_sig,} function_sig
function_sig	::=	identifier'('func_bindings')' **:** func_type
func_bindings	::=	{func_binding ,} func_binding
func_binding	::=	identifier **:** func_type
func_type	::=	type_name | **event_seq**
function_defs	::=	{function_def ,} function_def
function_def	::=	expr**==** expr
exprs	::=	{expr,} expr
inst_interfaces	::=	{inst_interface, } inst_interface
inst_interface	::=	interface_name['['type_names']']
type_names	::=	{type_name,} type_name

1.5 Names

interfaces	::=	{interface_name,} interface_name
classes	::=	{class_name,} class_name
interface_name	::=	identifier | **any**
class_name	::=	identifier
contract_name	::=	identifier
operation_name	::=	identifier
object_name	::=	id
type_name	::=	basic_type | identifier | (class_name | interface_name)**.**identifier
basic_type	::=	**int** | **nat** | **bool** | **string**
id	::=	identifier | id'('number')' | id**.**id
identifier	::=	non_num{alpha_num}
non_num	::=	**a** | ... | **z** | **A** | ... | **Z** | **_**
alpha_num	::=	digit | non_num | **-**
digit	::=	**0** | ... | **9**

1.6 Type Declarations

type_decls	::=	{*type_decl*,} *type_decl*
type_decl	::=	*identifier*: **type** = *type_expr*
type_expr	::=	*identifier*
	|	*enumeration_type*
	|	*tuple_type*
	|	*record_type*
	|	*seq_type*
	|	*set_type*
	|	*array_type*
	|	*subtype*
enumeration_type	::=	**'{'***identifiers***'}'**
tuple_type	::=	'['*tuple_members*']'
tuple_members	::=	{*tuple_member*,} *tuple_member*
tuple_member	::=	*identifier* **:** *type_expr*
record_type	::=	'['#*field_decls*#']'
field_decls	::=	{*field_decl*,} *field_decl*
field_decl	::=	*identifiers* **:** *type_expr*
seq_type	::=	**seq**'['*type_name*']'
set_type	::=	**set**'['*type_name*']'
array_type	::=	**array**'['*number*']''['*type_name*']'
subtype	::=	**'{'***binding mid expr***'}'**
identifiers	::=	{*identifier*,} *identifier*
mid	::=	´ | ´

1.7 Expressions

expr	::=	*litteral_expr*
	|	*event_sequence*
	|	*projection_set*
	|	*rs*
	|	*id*
	|	**'('***expr***')'**
	|	(*class_name* | *interface_name*).*identifier func_arguments*
	|	(*seq_pdf* | *id*) *func_arguments*
	|	*expr binop expr*
	|	*unaryop expr*
	|	*if_expr*
	|	*quantified_expr*
	|	*reused_spec*

litteral_expr ::= *number* | *boolean* | *string* |
'['*ex_exprs*']' | '{'*ex_exprs*'}' | '('*ex_exprs*')' | (#*ex_exprs*#) |
'['']' | '{''}' | '('')'

if_expr ::= **if** *expr* **then** *expr*
{**elsif** *expr* **then** *expr*}
else *expr* **endif**

quantified_expr ::= *binding_op bindings* **:** *expr*

func_arguments ::= '('{*expr*,} *expr*')'

binop ::= ⇔|⇒| ∨ | ∧ | **xor** | **andthen** | **orelse** |ˆ| + | − | /
| ∗ | % |=|≤|≥|<|>|≠|⊢|⊣|⊢⊣| **head** | \ | **prs** | **in** | **sub**

unaryop ::= **not** |<>| −

binding_op ::= **forall** | **exists**

reused_spec ::= (*class_name* | *interface_name*).**inv** |
(*class_name* | *interface_name*).**asm**

1.8 Communication events

rs ::= *event_sequence* | ´[´*rs mid* **where** *bindings*´]´ |
'('*rs*')' | *rs mid rs* | *rs** | *rs rs*

event_sequence ::= **empty** | {*event*} *event*

projection_set ::= *init* | *term* | *id* | *object_binding* |
event_set | *operation_set*

event_set ::= ´{´{*event*,}*event*´}´ | *event*

operation_set ::= ´{´{*operation_name*,}*operation_name*´}´ |
operation_name

init ::= →| *object* → *object*

term ::= ←| *object* ← *object*

init_term ::= ↔| *object* ↔ *object*

object ::= *object_name* | **me**

event ::= *init_event* | *term_event* | *init_term_event*

init_event ::= *init*.*operation_name*['('*exprs*')']

term_event ::= *term*.*operation_name*['('*exprs* [; *exprs*]')']

init_term_event ::= *init_term*.*operation_name*

1.9 Basic types

number ::= [-]{*digit*}*digit*

string ::= "{*ascii*}"

boolean ::= **true** | **false**

1.10 Imperative code

imperative_code ::= *stms*
stms ::= {*stm* **;**} *stm*
stm ::= *skip* | *if_stm* | *nondet_stm* | *while_stm* | *assignm* | *guarded_stm* | *ifany_stm* | *local_call* | *remote_call* | *local_var* | *mythical_stm*
skip ::= **skip**
if_stm ::= **if** *ex_expr* **then** *stms* [**else** *stms*] **endif**
nondet_stm ::= **begin** {*guarded_stm mid*} *guarded_stm* **end**
while_stm ::= **while** *ex_expr* **do** *stms* **enddo**
assignm ::= *ids* **:=** *ex_expr*
guarded_stm ::= *ex_expr* → *stm*
ifany_stm ::= **if any** *Itest_expr* **then** *stms* [**else** *stms*] **endif**
local_call ::= *operation_name*'('*ex_exprs*')' | *operation_name*'('*ex_exprs* **;** *ids*')'
remote_call ::= *identifier*.*local_call* [**do** *stms* **enddo**]
local_var ::= **var** *identifier* **:** *type_name* = *ex_expr*
mythical_stm ::= *bool_expr*
ids ::= {*id*,} *id*

1.11 Executable expressions

ex_exprs ::= {*ex_expr*, }*ex_expr*
ex_expr ::= *numeric_expr* | *bool_expr* | *string_expr* | *seq_expr* | *litteral_expr* | **null** | *id* | *new_expr* | '('*ex_expr*')'
bool_expr ::= *ex_expr bool_op ex_expr* | **not** *ex_expr* | *Itest_expr* | *id* | *boolean*
Itest_expr ::= *object_binding* **?**
numeric_exp ::= *ex_expr num_op ex_expr*
string_expr ::= *ex_expr* + *ex_expr*
seq_expr ::= *ex_expr seq_op ex_expr* | *seq_pdf*'('*ex_expr*')'
new_expr ::= **new** *class_name*'('*ex_exprs*')'
bool_op ::= $<|>|\leq|\geq|=|\neq|\wedge$ | **andthen** | $\vee$ | **orelse** | **xor** | **in** | **head** | **sub**
num_op ::= $+ \mid - \mid / \mid * \mid \%$
seq_op ::= $\vdash|\dashv|\vdash\!\dashv$
seq_pdf ::= **lr** | **rr** | **lt** | **rt** | #

1.12 Reserved words

andthen any asm axiom begin caller class contract Data_Type empty def do else elsif end endif enddo exists false forall func H if implements in inherits init Interface interface inv lma me new null opr orelse out skip super then true type types val var where while with

Types: array bool event_seq int nat seq set string
Functions and operators: head in lr lt not prs rr rt sub xor

1.13 ASCII symbols

Latex	*ASCII*
$\Leftrightarrow$	<=>
$\Rightarrow$	=>
$\vee$	\/
$\wedge$	/\
$\vdash$	| −
$\dashv$	− |
$\vdash\!\dashv$	| − |
$\leq$	<=
$\geq$	>=
$\neq$	/ =

1.14 Transformation

id ::= *identifier* [*id'*]
id' ::= **'('***number***')'** [*id'*] | **.***id* [*id'*]

expr ::= *litteral_expr* [*expr'*]
| *event_sequence* [*expr'*]
| *projection_set* [*expr'*]
| *rs* [*expr'*]
| *id* [*expr'*]
| **'('***expr***')'** [*expr'*]
| (*class_name* | *interface_name*)**.***identifier func_arguments* [*expr'*]
| (*seq_pdf* | *id*) *func_arguments* [*expr'*]
| *unaryop expr* [*expr'*]
| *if_expr* [*expr'*]
| *quantified_expr* [*expr'*]
| *reused_spec* [*expr'*]
expr' ::= *binop expr* [*expr'*]

rs ::= *event_sequence* [*rs'*] | **'['***rs mid* **where** *bindings***']'** [*rs'*] | **'('***rs***')'** [*rs'*]
rs' ::= *mid rs* [*rs'*] | ***** [*rs'*] | *rs* [*rs'*]

ex_expr ::= *litteral_expr* [*ex'*] | **null** [*ex'*] | *id* [*ex'*] | *new_expr* [*ex'*] |
'('*ex_expr***')'** [*ex'*] | *seq_pdf***'('***ex_expr***')'** [*ex'*] | *Itest_expr* [*ex'*] | **not** *ex_expr* [*ex'*]
ex' ::= *num_op ex_expr* [*ex'*] | *bool_op ex_expr* [*ex'*] | + *ex_expr* [*ex'*] |
seq_op ex_expr [*ex'*]

References

[1] The ADPAT-FT project homepage. http://www.ifi.uio.no/∼adapt/.

[2] Dahl, O.-J. *Verifiable Programming.* Prentice-Hall, 1992.

[3] Dahl, O.-J., and Owe, O. Formal methods and the RM-ODP. Tech. Rep. 261, Department of Informatics, University of Oslo, 1998.

[4] DAHL, O.-J., AND OWE, O. Formal Development with ABEL. In *Proceedings of Formal Software Development Methods (VDM '91) LNCS 552, pages 320–362*, 1991.

[5] OWE, O., AND RYL, I. OUN: A formalism for open, object-oriented, distributed sytems. Research Report 270, Department of Informatics, University of Oslo, Norway, 1999. http://www.ifi.uio.no/˜ adapt/.

[6] SOUNDARAJAN, N., AND FRIDELLA, S. Inheritance: From code reuse to reasoning reuse. In *Proc. 5th Conference on Software Reuse (ICSR5)* (1998), P. Devanbu and J. Poulin, Eds., IEEE Computer Society Press, pp. 206–215.