The syntax of the OUN language

Olaf Owe
Department of Informatics, University of Oslo, Norway

February 21, 2002

Contents

1 The OUN language 1
  1.1 Interface and contract definition 2
  1.2 Class and subclass definition 3
  1.3 Dynamic class extension 3
  1.4 Basic elements: Lists and bindings 4
  1.5 Names 4
  1.6 Type Declarations 5
  1.7 Expressions 5
  1.8 Communication events 6
  1.9 Basic types 6
  1.10 Imperative code 7
  1.11 Executable expressions 7
  1.12 Reserved words 7
  1.13 ASCII symbols 8
  1.14 Transformation 8

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 \( \text{guard } \rightarrow \text{statement} \), 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 \( m(\text{in}; \text{out}) \) do \( s \) od will execute the statements \( s \) 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 addition to class and interfaces specification, we allow contract specifications for subsystems involving 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

\[ spec_{\text{interface}} ::= \]
\[ \text{interface } \text{name} \left[ \left[ \text{type}\_\text{bindings} \right] \right] \left[ \left( \text{object}\_\text{bindings} \right) \right] \]
\[ \left[ \text{inherits } \text{interfaces} \right] \]
\[ \text{begin} \]
\[ \left[ \text{types } \text{type}\_\text{decls} \right] \]
\[ \left[ \text{with } \text{interface}\_\text{name} \right] \]
\[ \left\{ \text{operation} \right\} \]
\[ \left[ \text{asm } \text{expr} \right] \]
\[ \left[ \text{inv } \text{expr} \right] \]
\[ \left[ \text{auxiliary}\_\text{part} \right] \]
\[ \text{end} \]

\[ spec_{\text{contract}} ::= \]
\[ \text{contract } \text{name} \]
\[ \text{begin} \]
\[ \left[ \text{with } \text{object}\_\text{bindings} \right] \]
\[ \left[ \text{inv } \text{expr} \right] \]
\[ \left[ \text{auxiliary}\_\text{part} \right] \]
1.2 Class and subclass definition

\[ \text{spec\_class ::= \text{class}\ class\_name [\text{[ type\_binds]\]}[\text{[\('bindings')\]}]]} \]
\[ \text{implements\ inst\_interfaces} \]
\[ \text{inherits\ classes} \]
begin
\[ \text{types\ type\_decls} \]
\[ \text{val\ constant\_declarations} \]
\[ \text{var\ var\_declarations} \]
\[ \text{init\ imperative\_code} \]
\[ \{\text{operation == imperative\_code}\} \]
\[ \text{with\ interface\_name} \]
\[ \{\text{operation == imperative\_code}\} \]
\[ \{\text{asm\ expr}\} \]
\[ \text{inv\ expr} \]
\[ \text{auxiliary\ part} \]
end

1.3 Dynamic class extension

\[ \text{spec\_class\_extension ::= \text{class\ extension}\ class\_name} \]
\[ \text{implements\ inst\_interfaces} \]
begin
\[ \{\text{operation == imperative\_code}\} \]
\[ \text{with\ object\_binding} \]
\[ \{\text{operation == imperative\_code}\} \]
\[ \{\text{asm\ expr}\} \]
\[ \text{inv\ expr} \]
\[ \text{auxiliary\ part} \]
end

auxiliary\ part\ ::= \n\[ \text{func\ function\_sigs} \]
\[ \text{def\ function\_defs} \]
\[ \text{axiom\ exprs} \]
\[ \text{lma\ exprs} \]
1.4 Basic elements: Lists and bindings

\[
\begin{align*}
type \text{\_bindings} & ::= \{type\_binding,\} type\_binding \\
object \text{\_bindings} & ::= \{object\_binding,\} object\_binding \\
bindings & ::= \{binding,\} binding \\
type\_binding & ::= type\_name : type\_deter \\
object\_binding & ::= object\_name : interface\_name \\
binding & ::= identifier : type\_name \\
var\_declarations & ::= \{var\_declaration,\} var\_declaration \\
var\_declaration & ::= binding [:= expr] \\
constant\_declarations & ::= \{var\_declaration,\} var\_declaration \\
type\_deter & ::= Data\_Type | Interface \\
op \text{\_operation} & ::= opr \text{\_operation\_name}'([in\_out\_parameters])' \\
in\_out\_parameters & ::= [in] bindings [: out bindings] | out bindings \\
function\_sigs & ::= \{function\_sig,\} function\_sig \\
function\_sig & ::= identifier'[function\_bindings]' : func\_type \\
func\_bindings & ::= \{func\_binding,\} func\_binding \\
func\_binding & ::= identifier : func\_type \\
func\_type & ::= type\_name | event\_seq \\
function\_defs & ::= \{function\_def,\} function\_def \\
function\_def & ::= expr == expr \\
exprs & ::= \{expr,\} expr \\
inst\_interfaces & ::= \{inst\_interface,\} inst\_interface \\
inst\_interface & ::= interface\_name['[type\_names']'] \\
type\_names & ::= \{type\_name,\} type\_name
\end{align*}
\]

1.5 Names

\[
\begin{align*}
\text{interfaces} & ::= \{interface\_name,\} interface\_name \\
\text{classes} & ::= \{class\_name,\} class\_name \\
\text{interface\_name} & ::= identifier | any \\
\text{class\_name} & ::= identifier \\
\text{contract\_name} & ::= identifier \\
\text{operation\_name} & ::= identifier \\
\text{object\_name} & ::= id \\
\text{type\_name} & ::= basic\_type | identifier | \\
& \quad (class\_name | interface\_name),identifier \\
basic\_type & ::= int | nat | bool | string \\
id & ::= identifier | id'(number)' | id.id \\
identifier & ::= non_num\{alpha\_num\} \\
non_num & ::= a | \ldots | z | A | \ldots | Z \\
alpha\_num & ::= digit | non_num | - \\
digit & ::= 0 | \ldots | 9
\end{align*}
\]

4
1.6 Type Declarations

\[
\begin{align*}
type\_decls & ::= \{type\_decl,\} type\_decl \\
type\_decl & ::= identifier \textbf{type} = type\_expr \\
type\_expr & ::= \text{identifier} \\
& \quad \mid \text{enumeration\_type} \\
& \quad \mid \text{tuple\_type} \\
& \quad \mid \text{record\_type} \\
& \quad \mid \text{seq\_type} \\
& \quad \mid \text{set\_type} \\
& \quad \mid \text{array\_type} \\
& \quad \mid \text{subtype}
\end{align*}
\]

\[
\begin{align*}
\text{enumeration\_type} & ::= \{\text{"identifiers"}\}' \\
\text{tuple\_type} & ::= \text{"["tuple\_members"]"} \\
\text{tuple\_members} & ::= \{\text{tuple\_member},\} \text{tuple\_member} \\
\text{tuple\_member} & ::= \text{identifier} : type\_expr \\
\text{record\_type} & ::= \text{"["#field\_decls"#"]"} \\
\text{field\_decls} & ::= \{\text{field\_decl},\} \text{field\_decl} \\
\text{field\_decl} & ::= \text{identifiers} : type\_expr \\
\text{seq\_type} & ::= \text{seq}["type\_name"]' \\
\text{set\_type} & ::= \text{set}["type\_name"]' \\
\text{array\_type} & ::= \text{array}["number"]["type\_name"]' \\
\text{subtype} & ::= \{\text{"binding mid expr"}\}' \\
\text{identifiers} & ::= \{\text{identifier},\} \text{identifier} \\
\text{mid} & ::= \text{"\mid"}
\end{align*}
\]

1.7 Expressions

\[
\begin{align*}
expr & ::= \text{litteral\_expr} \\
& \quad \mid \text{event\_sequence} \\
& \quad \mid \text{projection\_set} \\
& \quad \mid \text{rs} \\
& \quad \mid \text{id} \\
& \quad \mid \{\text{expr}\}' \\
& \quad \mid (\text{class\_name} | \text{interface\_name})\text{identifier} \hspace{1pt} \text{func\_arguments} \\
& \quad \mid (\text{seq\_pdf} | \text{id}) \hspace{1pt} \text{func\_arguments} \\
& \quad \mid \text{expr} \hspace{1pt} \text{binop} \hspace{1pt} \text{expr} \\
& \quad \mid \text{unaryop} \hspace{1pt} \text{expr} \\
& \quad \mid \text{if\_expr} \\
& \quad \mid \text{quantified\_expr} \\
& \quad \mid \text{reused\_spec}
\end{align*}
\]
```
literal_expr ::= number | boolean | string |
              | ['"ex_exprs"'] | '{ex_exprs}' | '('ex_exprs')' | (#ex_exprs#) |
              | ']' | '}' | '(')
if_expr ::= if expr then expr
           {elseif expr then expr} else expr endif
quantified_expr ::= binding_op bindings : expr
func_arguments ::= ('{exprs} expr')
binop ::= $|\triangledown|\bigtriangledown | xor | andthen | orelse | + | - | /
         | * | % | =|<=$>>| |>|<$|=|<|=|>|=|>|
unaryop ::= not |<>| -
func_arguments ::= 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

operation_set ::= '{operation_name,operation_name}' |
                 | operation_name

init ::= \rightarrow object \rightarrow object
term ::= \leftarrow object \leftarrow object
init_term ::= \leftrightarrow object \leftrightarrow object

object ::= object_name | me

init_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}"n
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 ltest 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 |
literal_expr | null | id | new expr | ('ex expr')
bool_expr ::= ex expr bool op ex expr | not ex expr | ltest expr |
| id | boolean
ltest expr ::= object binding ?
numeric_expr ::= 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 ::= < | <= | >= | != | ~ | andthen | | | orelse | xor |
in | head | sub
num_op ::= + | − | / | * | %
seq_op ::= − | | =
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

<table>
<thead>
<tr>
<th>Latex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Leftrightarrow$</td>
<td>$\leftrightsquigarrow$</td>
</tr>
<tr>
<td>$\Rightarrow$</td>
<td>$\Rightarrow$</td>
</tr>
<tr>
<td>$\land$</td>
<td>$\land$</td>
</tr>
<tr>
<td>$\lor$</td>
<td>$\lor$</td>
</tr>
<tr>
<td>$\vdash$</td>
<td>$\vdash$</td>
</tr>
<tr>
<td>$\not\vdash$</td>
<td>$\not\vdash$</td>
</tr>
</tbody>
</table>

1.14 Transformation

\[
\begin{align*}
\text{id} & ::= \text{identifier} \ [id] \\
\text{id}' & ::= \text{'(number')} \ [id'] \ | id \ [id'] \\
\text{expr} & ::= \text{litteral_expr} \ [expr'] \\
& \quad \text{| event_sequence} \ [expr'] \\
& \quad \text{| projection_set} \ [expr'] \\
& \quad \text{| rs} \ [expr'] \\
& \quad \text{| id} \ [expr'] \\
& \quad \text{| '(expr')'} \ [expr'] \\
& \quad \text{| (class_name | interface_name).identifier func_arguments} \ [expr'] \\
& \quad \text{| (seq_pdf | id) func_arguments} \ [expr'] \\
& \quad \text{| unaryop expr} \ [expr'] \\
& \quad \text{| if_expr} \ [expr'] \\
& \quad \text{| quantified_expr} \ [expr'] \\
& \quad \text{| reused_spec} \ [expr'] \\
\text{expr'} & ::= \text{binop expr} \ [expr'] \\
\text{rs} & ::= \text{event_sequence} \ [rs'] \ | '[rs mid where bindings']' \ [rs'] \ | '(rs')' \ [rs'] \\
\text{rs'} & ::= \text{mid rs} \ [rs'] \ | * \ [rs'] \ | rs \ [rs'] \\
\text{ex_expr} & ::= \text{litteral_expr} \ [ex'] \ | \text{null} \ [ex'] \ | \text{id} \ [ex'] \\
& \quad \text{| new_expr} \ [ex'] \ | \text{'(expr')'} \ [ex'] \\
& \quad \text{| seq_pdf} \text{'}(\text{ex_expr})' \ [ex'] \\
& \quad \text{| if_expr} \ [ex'] \\
& \quad \text{| not_expr} \ [ex'] \\
\text{ex'} & ::= \text{num_op ex_expr} \ [ex'] \ | \text{bool_op ex_expr} \ [ex'] \\
& \quad \text{| + ex_expr} \ [ex'] \\
& \quad \text{| seq_op ex_expr} \ [ex']
\end{align*}
\]

References

