Grammar of the New Programming Language

David Talby

Institute of Computer Science
The Hebrew University, Jerusalem, Israel
davidt@cs.huji.ac.il

Abstract. The concept of Contract-Oriented Programming, or the use of assertions in a program's source code at compile time, has two main benefits. The first is allowing many more compile-time tests of bugs which were impossible to detect statically before. The second is the ability to greatly simplify the programming language by automating decisions that are undecidable at compile time without the use of assertions. The New programming language is the first one designed to take advantage of these abilities, and its formal grammar is given here. It is based on a pure object-oriented language, supporting multiple inheritance, genericity and dynamic dispatch based on types, exports or predicates. Its major strengths are the automation of thread and process creation, lock assignment to shared objects, scheduling and rollback-based exception handling.

1. Rational

Software faults are nowadays the most serious problem of the computing and communication industries. Software is in fact the only industry in which products which passed every existing testing method and quality standard crash on a daily basis. The reason for this is that there are several classes of bugs which are extremely difficult for humans to track using common debuggers. Among these are memory bugs (leaks, using uninitialized objects, using destroyed objects), error handling bugs (ignoring errors, not restoring objects to a stable state), concurrency bugs (race conditions, deadlock, starvation), time related bugs (cumulative drift) and others.

The only viable option is to find these errors using an automatic tool. Compilers include many such tests for common errors – misspelled words, ambiguities, and type errors to name a few. However, the errors mentioned above are very resistant to such treatment: Tracking them is an undecidable problem in general, and impossible even in very short and simple programs in practice. While modern compilers discover most of the unreachable code and unused variables in programs – two other theoretically undecidable problems – that occur in practice, they are helpless when faced, for example, with a concurrency bug.

In short – automatic detection is impossible, manual detection doesn't work, and our software sure looks that way.

The most widespread method of dealing with these issues is automation of the problematic issues. Automatic garbage collection, for example, makes all memory related errors except the use of uninitialized objects go away. Guaranteed rollbacks by transaction managers provide safe error handling, and some systems try to offer automatic parallelization that is safe from common concurrency problems. All of these are very welcome; beyond preventing bugs, they drastically cut development time and cost of complex systems. Their only problem is performance. Backing up all data before a computation to prepare for a potential rollback is acceptable for critical database applications, but not for numerical programs or games. Even the moderate cost of modern garbage collectors is too expensive for some applications. As a result, “general purpose” programming languages usually don’t include such automatic facilities, and actually take pride in giving the programmer full manual control.

Contract-Oriented Programming enters the picture here to solve both problems at once: Test at compile time for undecidable “elusive” bugs, and enable more efficient automatic facilities. The basic idea is simple. Since the source code by itself is not enough to make a decision, the programmer will augment the code with assertions. New compile-time algorithms will rely on these assertions for both error checking and automation. The trick is that the assertions themselves can be checked, at runtime, and optionally not checked, which removes any runtime overhead the checking may have.

Assertions in software are not a new idea. They have a sound theoretical basis in the theory of abstract data types, whose best translation to practice is the concept of “Design by Contract” [2] of the Eiffel language. Every routine defines a contract towards its clients, in the form of a precondition (what it requires) and a postcondition (which it ensures assuming the precondition holds), and every class defines a contract in terms of its invariant. For example, a square root function would be defined:

```
sqrt(x: Real): Real
    require
        x >= 0
    do
        … actual code here …
    ensure
        result * result = x
    end
```

The goals of Design by Contract are to support the design, testing and documentation of programs. Eiffel programmers indeed report it to be of great help in those areas. However, the idea has not spread to other languages and is not popular, mainly because assertions have no “real” effect on the code, and many programmers just don’t find the time to add them worth their while. This is the single most serious problem of Design by Contract: It aids every stage of a software project’s life cycle except coding, which kept it out of sight of most development environments, which are centered around their compiler.

However, by using assertions at compile-time the contracts become a vital part of the language’s semantics. Not only that new errors are caught (so assertions must be placed to “calm” the compiler), the contracts are also used to define the program’s behavior. When you assert that an object is private, you actually program is memory deallocation scheme. When you assert that two pointers may point to the same object, you’ve just requested a semaphore. Every aspect of the code relies on the assertions around it. This is the heart of the shift from contracts in the language to a contract-oriented language: The assertions are an integral part of the code.

Some of the ideas and algorithms are applicable to existing programming languages, but the concept really calls for the design of a new one. First, since it is possible to automate memory, multi-threading, mutual exclusion, scheduling, rollbacks and dynamic dispatch efficiently, many mechanisms in existing languages will be redundant. A far simpler design is possible. Second, it is interesting to test how new and advanced mechanisms that only few languages support (like predicate dispatch or dynamic class loading) fit into the scheme and interact with other mechanisms. Third, an important factor in the success of contract-orientation is the ability to learn as much as possible from few assertions, and avoid polluting the code with too many of them. The language should be designed so that extracting information from the source code is as easy as possible.

The New programming language, whose grammar fills the remainder of this paper, is the first result of these three arguments. It tries to fulfill all three promises, and includes several new mechanisms and compiler algorithms. For a detailed analysis of each facility and comparison with other approaches, see [1].

2. Conventions

The conventions used to describe the syntax are based on [3]:

- Keywords such as **inherit** and symbols such as **:=** are printed in bold.
- Non-terminals such as *Creation_clause* are printed in Italics.
- Each non-terminal appears once at the beginning of a production, which describes to what other lexical elements it should be broken:
  Conditional
      if *Then_part_list* [*Else_part*] **end**
- Optional constructs, such as *Else_part* above, are enclosed in brackets [].
- Choices are printed by separating the variants by the | symbol:
  Feature_Name
      Identifier | *Operator*
- Repetitions of a variable number of times of the same constructed are printed in one of two forms:
  { *Construct Separator* … }
  { *Construct Separator* … } +

  Both forms allow a variable number of constructs, separated by the separator if more than one is used. In the first form there may be zero constructs, but in the second one at least one is required.
- When a comma is used as a separator in a repetitive production, then a semi-colon or a new-line character can be used as well. This is just a shorthand for defining the following non-terminal:
  List_separator
      , | **;** | *New_line*
- Terminal tokens such as *Integer* and *Identifier* are described in English.
- Comments may be used freely everywhere. However, expected comments such as those describing the formal argument of a routine or a class, or those describing a routine as obsolete or unbounded, are part of the official syntax.
- Notes about rules which can be enforced by the grammar but are not so for ease of presentation are given in separate lines, starting with the word <u>*Note:*</u>

3. The Syntax

Class_declaration
Class_header
[*Formal_generics*]
[*Class_header_comments*]
[*Inheritance*]
[*Creators*]
[*Features*]
[*Invariant*]
end

Class_Header
[*Header_mark*] **class** *Class_name*
Header_mark
Entity_kind **|** **abstract**
Entity_kind
reference | value | free
Class_name
Identifier
Formal_generics
[*Formal_generics_list* **]**
Formal_generics_list
{ *Formal_generic*, … } +
Formal_generic
Formal_generic_name [*Constraint*]
Formal_generic_name
Identifier
Constraint
-> *Class_name* | = *Size_Constant*
Size_constant
Integer_constant | **unbounded**

Class_header_comments
[*Class_description_list*]
[*Formal_generics_description_list*]
[*Obsolete_comment*]
Class_description_list
{ *Comment* … } +
Formal_generics_description_list
{ *Formal_generic_description* … } +
Formal_generic_description
Formal_generic **:** *Comment*
Obsolete_comment
obsolete : *Comment*
Comment
-- *String New_line*
Header_comment
Comment

Inheritance
inherit { *Inherit_clause* **inherit** } +
Inherit_clause
[*Clients*] [*Header_comment*] *Parent_list*
Parent_list
{ *Parent* **,** … } +
Parent
Class_name [*Feature_Adaptation*]
Feature_Adaptation
[*Rename*]
[*Redefine*]
[*Rebind*]
end

Note: The **end** keyword can only be used here if at least one of the three optional constructs is used. Otherwise the syntax is not LR(1).

Rename
rename *Rename_list*
Rename_list
{ *Feature_pair* , … } +
Feature_pair
Feature_name **as** *Feature_name*

Redefine
redefine *Redefine_List*
Redefine_List
{ *Redefine_mark* , … } +
Redefine_mark
Feature_name | *Rename_pair* | *Join_Feature_pair*
Join_Feature_pair
Feature_name **as** *Classed_Feature_name*
Classed_Feature_name
Class_name **.** *Feature_name*

Rebind
rebind *Rebind_list*
Rebind_list
{ *Rebind_mark* , … } +
Rebind_mark
May_pair | *Must_pair* | *May_not_pair* | *Must_not_pair* | *Feature_name*
Any_May_pair
Feature_name **may** *Any_Feature_name*
Any_Must_pair
Feature_name **must** *Any_Feature_name*
Any_May_not_pair
Feature_name **may not** *Any_Feature_name*
Any_Must_not_pair
Feature_name **must not** *Any_Feature_name*
Any_Feature_Name
Feature_name | *Classed_Feature_name*

Creators
  new { *Create_clause* **new** … } +
Create_clause
  [*Clients*] [*Header_comment*] Commands_list
Commands_list
  { *Feature_name* , … }
<u>*Note:*</u> Only commands (i.e. procedures) may be used as creators.

Features
  [*Feature_list_mark*] **feature** { *Feature_clause* **feature** … } +
Feature_list_mark
  abstract | **final**
Feature_clause
  [*Clients*] [*Header_comment*] *Feature_list*
Feature_list
  { *Feature_declaration* , … }

Feature_declaration
  Feature_name_list Declaration_body
Feature_name_list
  { *Feature_name* , … } +
Feature_name
  Identifier | **set** *Identifier* | **operator** *Definable_operator*
Definable_operator
  Unary_operator | *Binary_operator* | **new** | **( )**
Declaration_body
  [*Formal_arguments*] [*Type_mark*] [*Constant_or_routine*]
Constant_or_routine
  := *Constant_value* | *Routine*
Constant_value
  Manifest_constant | **constant**

Formal_arguments
  (*Arguments_list* **)**
Arguments_list
  { *Arguments_group* , … } +
Arguments_group
  [**set**] *Entity_list Type_mark*
Entity_list
  { *Entity* , … } +
Entity
  Identifier
Type_mark
  : *Type*

Routine
[*Routine_header_comments*] [*May_variants*] [*Must_variants*]
[*Precondition*] [*Local_entities*] [*Loop_counters*]
[*Routine_body*] [*Postcondition*] [*Catch_clauses*] **end**

Routine_header_comments
[*Routine_description_list*]
[*Formal_arguments_description_list*]
[*Grammar_comments_list*]
Routine_description_list
{ *Comment* … } +
Formal_arguments_description_list
{ *Formal_generic_description* … } +
Formal_argument_description
Formal_argument **:** *Comment*
Grammar_comment
Obsolete_comment | *Throw_comment* | *Guard_comment* |
unbounded | **robust** | **nonreversible**
Throw_comment
throw : *Type_list*
Guard_comment
guard : *Identifier_list*
Note: "robust", "guard" and "nonreversible" are not keywords, yet they have a fixed meaning. The same goes for the special comments "usually", "rollback" and "delete" in routine bodies.

May_variants
may { *Feature_name* , … } +
Must_variants
must { *Feature_name* , … } +

Precondition
require [**else**] *Assertions*
Postcondition
ensure [**then**] *Assertions*
Invariant
invariant *Assertions*
Assertions
{ *Assertion* , … } +
Assertion
[*Tag_mark*] *Unlabeled_assertion*
Tag_mark
Identifier **:**
Unlabeled_assertion
Comment | *Boolean_expression*

Local_entities
local { *Entities_group*, … } +
Entities_group
Entity_list Type_mark

Routine_body
do *Compound*

Catch_clauses
catch { *Catch_clause* **catch** … } +
Catch_clause
[*Clients*] [*Header_comment*] *Compound*

Compound
{ *Instruction* , … }
Instruction
Call | *Assignment* | *Condition* | *For* | *While* | *Allow* | *Assert* | *Throw* | *Retry* | *(Null)*

Assignment
Writable **:=** *Expression*
Writable
result | *Entity*

Condition
if Then_part_list [Else_part] **end**
Then_part_list
[*Then_part* **elseif** …] +
Then_part
Boolean_expression **then** *Compound*
Else_part
else *Compound*

For
for *Iterators_list* **loop** *Compound* **end**
Iterators_list
{ *Named_iterator* , … }
Named_iterator
Identifier **in** *List_or_iterator*
List_or_iterator
List_expression | *Iterator_expression*

While
while *Boolean_expression Loop_counters* [Loop_invariant] **loop** *Compound* **end**
Loop_counters
from *Size_constant* **to** *Size_constant*
Loop_invariant
invariant *Assertions*

Allow
allow [**from** *Expression*] [**to** *Expression*] *Allow_mark Compound* **end**
Allow_mark
for | **then**

Assert
assert *Boolean_expression*

Throw
    throw *Expression*
Note: The thrown expression must be (a descendant of) class Exception.

Retry
    retry
Note: The retry instruction can only appear inside a catch clause.

Call
    Quantified_call | *Super_call* | *External_call*

Quantified_call
    [*Call_target* **.**] *Call_chain*
Call_target
    Parenthesized | *New* | *Super* | **this** | **result**
Call_chain
    { *Unqualified_call* **.** … } +
Unqualified_call
    Identifier [*Actuals*]
Actuals
    (*Actuals_list* **)**
Actuals_list
    { *Expression* , … } +
Super
    [*Class_name* **.**] **super**

Super_call
    Super [*Actuals*]

External_call
    external " [*Language_name*] *External_name* **"** [*Actuals*]
Language_name
    String
External_name
    String

Type
    Class_type | *Tuple_type* | *Anchored_type*
Class_type
    [*Entity_kind*] *Class_name* [*Actual_generics*]
Actual_generics
    [*Type_list* **]**
Type_list
    { *Type* , … } +
Tuple_type
    [*Type_list* **]**
Anchored_type
    like *Anchor*
Anchor
    this | *Entity*

Expression
Container_expression | Manifest_constant | **this** | **result** | *Call* |
Operator_expression | *Equality* | *New* | *Old* | *Dynamic_cast*

Boolean_expression
Expression
Note: The expression must (obviously) be of type boolean.

Container_expression
Integer_sequence | *Tuple_expression* | *Set_expression*
Integer_sequence
Expression **..** *Expression*
Tuple_expression
[*Expression_list*]
Set_expression
{ *Expression_list* }
Expression_list
{ *Expression* , … }
Note: The expression in an integer sequence must be of type integer. The expressions that form a tuple or set do not have to be of the same type.

Operator_expression
Parenthesized | *Unary_expression* | *Binary_expression*
Parenthesized
(*Expression* **)**
Unary_expression
Prefix_operator Expression
Binary_expression
Binary_operator Expression
Unary_operator
+ | – | **not**
Binary_operator
+ | – | * | / | // | % | ^ | > | < | >= | <= |
in | **and** | **or** | **xor** | **implies**

Equality
Expression Comparison Expression
Comparison
= | **/=**

New
new *Class_type* [*Creator_call*]
Creator_call
. *Feature_name* [*Actuals*]

Old
old *Expression*

Dynamic_cast
Expression **as** *Class_type*

Manifest_constant
    Manifest_Integer | Real_constant |
    Boolean_constant | Character_constant | String_constant

Manifest_integer
    Integer_constant | Bit_constant | Hexadecimal_constant | Octal_constant

Integer_constant
    [*Sign*] *Integer*
Sign
    – | +
<u>Note:</u> This introduces an ambiguity into the grammar: is –3 an *Integer_constant* or an *Unary_expression*? The parser should choose the first alternative.

Bit_constant
    Bit_letter_list **b**
Bit_letter_list
    { *Bit_letter* … } +
Bit_letter
    0 | 1
<u>Note:</u> It is also legal to use a capital B in *Bit_constant* as in 0101B. Same for hex and octal.

Hexadecimal_constant
    Hex_letter_list **h**
Hex_letter_list
    { *Hex_letter* … } +
Hex_letter
    0-9 | A-F | a-f

Octal_constant
    Hex_letter_list **o**
Octal_letter_list
    { *Hex_letter* … } +
Octal _letter
    0-7

Integer
    { *Digit* … } +
Digit
    0-9

Real_constant
    [*Sign*] *Real*

Real
    Integer **.** *Integer* [*Exponent*]
Exponent
    e [*Sign*] *Integer*

<u>Note:</u> No intervening characters (such as spaces) are allowed inside a numeric constant.

Boolean_constant
true | **false**

Character_constant
' *Character* **'**
Character
(any printable character except \ or a special character with the same value as in C)

String_constant
" *String* **"**
String
{ *Character* … }

Identifier
Letter { *Alpha* … }
Letter
A-Z | **a-z**
Alpha
Letter | *Digit* | _

Notes:

- The grammar is case sensitive. However, it is illegal to declare two identifiers that only defer by case (such as *Aba* and *aba*).
- Comments start with two dashes -- and extend until the end of the line.
- Wherever a list is expected in the grammar, its elements can be separated by a new-line, a comma or a semicolon.
- A break (one of space, tab, new-line or carriage return) can be legally inserted between any two lexical elements except manifest constants.
- The compiler should know these basic classes: Integer, Real, String, Boolean, Any, None, System, List, Tuple, Array, Iterator.
- It is illegal to declare identifiers whose name equals that of a reserved word:
  if then else elseif end while for loop allow local do throw catch retry
  class feature abstract reference value free external may must operator set like
  from to invariant assert require ensure inherit rename redefine rebind final
  in and or xor not implies old new as
  true false result this unbounded super null constant runtime system
- This is the table of operator precedence. Within the same level computation is from left to right, except for the ^ (power) operator which is right-associative.

Feature calls:	. ( ) ?
All prefix operators:	**as**, **old**, **new**, **not**, unary + –
Power:	^
Multiplication and division:	* / // *(integer division)* % *(modulus)*
Addition and subtraction:	+ –
Interval list constructor:	.. (two dots)
Comparison:	= /= *(not equal)* < > <= >= **in**
Logical/Bitwise and:	**and**
Logical/Bitwise or:	**or** **xor**
Logical implication:	**implies**
Manifest collections:	[ ] *(tuples)* { } *(sets)* " " *(strings)*
List separators:	, ;

References

1. David Talby, “Contract-Oriented Programming”. Master Thesis, Hebrew University of Jerusalem, 1999.
2. Bertnard Meyer, “Object Oriented Software Construction, 2nd Edition”. Prentice Hall, 1997, pages 331-410.
3. Eric Bezault, “Eiffel: The Syntax”. http://www.gobosoft.com/eiffel/syntax/