First Draft of the act Programming Language

Eleftherios Matsikoudis
Christos Stergiou

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-5
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-5.html

January 22, 2014

Acknowledgement

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards \#0720882 (CSR-EHS: PRET), \#0931843 (CPS: Large: ActionWebs), and \#1035672 (CPS: Medium: Timing Centric Software)), the Naval Research Laboratory (NRL \#N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota.

First Draft of the act Programming Language*

Eleftherios Matsikoudis and Christos Stergiou

Abstract

The purpose of this report is to document the first complete attempt at the design of a high-level programming language for timed systems called act. We define the lexical grammar and the syntactic grammar of act, and include an example of a simple actor written in act, demonstrating some of the syntactic features of the language.

1 Introduction

The purpose of this report is to document the first complete attempt at the design of a high-level programming language for timed systems called act. act is a timed actor-oriented programming language. It is statically typed, featuring type inference and both parametric and ad hoc polymorphism. An act program starts with the execution of the actor **main**. **main** can create other actors to form a dynamically evolving network of conceptually concurrent, memory-isolated entities that communicate solely through message passing. All actors in a program share a global notion of logical time, which is a first-class citizen in the language, directly accessible via the keyword **time**. The language allows for polymorphism in the type of **time**. Logical time advances through the use of temporal statements such as **wait** and **whenever**. Non-temporal statements execute in zero logical time.

act is first and foremost a discrete-event modelling and simulation language. By relating logical to physical time, according to the PTIDES paradigm (see [4], [1]), it can also be used to program real-time systems. The algorithmic approach introduced in [3] can be extended to suitably chosen fragments of the language to perform the schedulability analysis necessary for hard real-time applications.

We use the Extended BNF syntactic metalanguage (see [2]) to define the lexical grammar (see Section 2) and the syntactic grammar (see Section 3) of act. Listing 1 shows the source code of a simple actor written in act, demonstrating some of the syntactic features of the language.

This report is by no means an introduction to act. It is simply a blueprint that is meant to record a starting point for the language, mostly for future reference and comparison purposes. The syntax reported here is largely the result of an intense period of work that took place between the summer of 2012 and the beginning of 2013. Since then, it has been extended and refined in various ways, but remains only a draft. There are more than a few omissions and ambiguities in it, and we expect the details of the language to be revised considerably as the design becomes informed by implementation and usage. Nevertheless, we think that even in its present form, this draft carries plenty of information about the basic ideas underlying the design of act, and will be useful in the future in understanding the evolution of these ideas and the language as a whole.

* This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS: PRET), #0931843 (CPS: Large: ActionWebs), and #1035672 (CPS: Medium: Timing Centric Software)), the Naval Research Laboratory (NRL #N0013-12-1-G015), and the following companies: Bosch, National Instruments, and Toyota.

Listing 1: The time-interval actor.

```
/** The time-interval actor measures the time interval between any two
 *  consecutive input events, and produces it at the output at the time the
 *  second of the two is received.
 */

actor TimeInterval<T, V> {
    interface {
        configure_input : private channel (& channel V)
                          access {
                              main    write
                          }
        out             : output channel T
    }

    timer : T

    thread() {
        helper = create Helper()

        wait helper.request? in [time, ...)
            timer = time

        whenever helper.request? in (time, ...) {
            out = time - timer
            timer = time
        }
    }

    actor Helper {
        interface {
            request : output channel V
        }

        input_request : & channel V

        thread() {
            configure()

            interrupt {
                whenever *input_request? in [time, ...)
                    request = *input_request
            } handle configure_input? in (time, ...) {
                 configure()
                 resume
            }
        }

        function configure() : unit = {
            whenever configure_input? in [time, ...) {
                if configure_input != null {
                    input_request = configure_input
                    break
                }
            }
        }
    }
}
```

2 Lexical grammar

Programs are written using the Unicode character set encoded in UTF-8. Streams of Unicode characters are translated into sequences of white space elements, comments, and the tokens of the syntactic grammar. Here, we are concerned with the lexical grammar of the latter.

2.1 Tokens

```
token = identifier
      | keyword
      | literal
      | operator or separator;
```

2.2 Identifiers

```
identifier = identifier string - (keyword | absent literal | boolean literal | null literal);

identifier string = letter, {letter | digit};

letter = "_" | unicode letter;

digit = unicode digit;
```

2.3 Keywords

```
keyword = "abort"       | "abstract"    | "access"      | "actor"       | "and"
        | "bool"        | "break"       | "buffer"      | "byte"        | "case"
        | "channel"     | "char"        | "constant"    | "continue"    | "create"
        | "default"     | "do"          | "else"        | "exit"        | "float"
        | "for"         | "from"        | "function"    | "group"       | "handle"
        | "if"          | "import"      | "in"          | "include"     | "infer"
        | "input"       | "int"         | "interface"   | "interrupt"   | "intersection"
        | "is"          | "literal"     | "main"        | "make"        | "match"
        | "new"         | "output"      | "package"     | "parallel"    | "private"
        | "product"     | "read"        | "record"      | "resume"      | "return"
        | "self"        | "skip"        | "string"      | "switch"      | "subtype"
        | "sum"         | "thread"      | "time"        | "type"        | "union"
        | "variant"     | "wait"        | "whenever"    | "while"       | "write";
```

2.4 Literals

```
literal = absent literal
        | boolean literal
        | byte literal
        | character literal
        | floating literal
        | integer literal
        | null literal
        | string literal;
```

```
absent literal = "absent";

boolean literal = "true" | "false";

byte literal = binary byte literal
             | hexadecimal byte literal;

binary byte literal = "0b", 8 * binary digit;

binary digit = "0" | "1";

hexadecimal byte literal = "0x", 2 * hexadecimal digit;

hexadecimal digit = decimal digit
                  | "a" | "b" | "c" | "d" | "e" | "f"
                  | "A" | "B" | "C" | "D" | "E" | "F";

decimal digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

character literal = "'", unicode value, "'";

unicode value = unicode code point - unicode line terminator
              | escape sequence;

escape sequence = octal escape sequence
                | hexadecimal escape sequence
                | little unicode escape sequence
                | big unicode escape sequence
                | simple escape sequence;

octal escape sequence = "\", 3 * octal digit;

octal digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7";

hexadecimal escape sequence = "\x", 2 * hexadecimal digit;

little unicode escape sequence = "\u", 4 * hexadecimal digit;

big unicode escape sequence = "\U", 8 * hexadecimal digit;

simple escape sequence = '\"' | "\'" | "\\" | "\a" | "\b" | "\f" | "\n" | "\r" | "\t" | "\v";

floating literal = decimal floating literal
                 | hexadecimal floating literal;

decimal floating literal = decimal fractional literal, [decimal exponent part]
                         | decimal digit sequence, decimal exponent part;

decimal fractional literal = decimal digit sequence, ".", decimal digit sequence;

decimal digit sequence = decimal digit, {decimal digit};

decimal exponent part = "e", [sign], decimal digit sequence;

sign = "+" | "-";

hexadecimal floating literal = "\0x", hexadecimal fractional literal, [binary exponent part]
                             | "\0x", hexadecimal digit sequence, binary exponent part;

hexadecimal fractional literal = hexadecimal digit sequence, ".", hexadecimal digit sequence;

binary exponent part = "p", [sign], decimal digit sequence;

hexadecimal digit sequence = hexadecimal digit, {hexadecimal digit};
```

```
integer literal = decimal literal
                | octal literal
                | hexadecimal literal;

decimal literal = non-zero digit, {decimal digit};

non-zero digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";

octal literal = "0", {octal digit};

hexadecimal literal = "\0x", hexadecimal digit, {hexadecimal digit};

null literal = "null";

string literal = interpreted string literal
               | uninterpreted string literal;

interpreted string literal = '"', {unicode value}, '"';

uninterpreted string literal = "`", {unicode code point}, "`";
```

2.5 Operators and separators

```
operator or separator = "!"    | "%"    | "%="   | "&"    | "&&"   | "&&&"  | "&&="  | "&="
                      | "("    | ")"    | "*"    | "*="   | "+"    | "++"   | "+="   | ","
                      | "-"    | "--"   | "-="   | "."    | "..."  | "/"    | "/="   | ":"
                      | ";"    | "<"    | "<<"   | "<<<"  | "<="   | "="    | "=="   | ">"
                      | ">="   | ">>"   | ">>>"  | "?"    | "@"    | "["    | "]"    | "^"
                      | "^="   | "{"    | "|"    | "|="   | "||"   | "||="  | "|||"  | "}"
                      | "~";
```

3 Syntactic grammar

A program consists of one or more so-called “translation units” stored in separate files. After lexical translation, these translation units are reduced to sequences of tokens. Here, we define their syntactic grammar.

3.1 Translation units

```
translation unit = {translation unit item};

translation unit item = definition
                      | import directive
                      | package declaration;

import directive = "import", import item, "from", package name;

import item = "*"
            | identifier;

package declaration = "package", package name;
```

```
package name = identifier, {".", identifier};
```

3.2 Definitions

```
definition = actor definition
           | constant definition
           | function definition
           | group definition
           | type definition;

actor definition = "actor", "main", actor block
                 | "actor", identifier, [type parameters], actor block;

actor block = "{", {actor block item}, "}";

actor block item = declaration
                 | definition
                 | include directive
                 | interface
                 | thread;

include directive = "include", identifier, [type arguments];

type arguments = "<", type argument list, ">";

type argument list = type argument, {",", type argument};

type argument = type name;

interface = [interface access attribute], "interface", interface block
          | "private", "interface", interface block,  [interface access control list];

interface access attribute = "input" | "output";

interface block = "{", {interface block item}, "}";

interface block item = uninitialized declaration;

interface access control list = "access", interface access control list block;

interface access control list block = "{", {interface access control entry}, "}";

interface access control entry = signature, interface access right;

signature = actor signature
          | function signature
          | group signature;

actor signature = "actor", "main"
                | "actor", identifier, [type arguments];

function signature = "function", identifier, [type arguments], ["(", [type name list],
                     ")", [":", type name]]
                   | "function", "(", overloadable operator, ")", ["(", [type name list],
                     ")", [":", type name]];

overloadable operator = "%"    | "%="   | "&"    | "&="   | "*"    | "*="   | "+"    | "++"
                      | "+="   | "-"    | "--"   | "-="   | "/"    | "/="   | "<"    | "<<<"
                      | "<<<=" | "<="   | ">"    | ">="   | ">>>"  | ">>>=" | "^"    | "^="
                      | "|"    | "|="   | "~";
```

```
group signature = "group", identifier, [type arguments];

interface access right = "read" | "read", "/", "write" | "write";

thread = "thread", parameters, compound expression;

parameters = "(", [parameter list], ")";

parameter list = parameter, {",", parameter};

parameter = identifier, [":", type name], ["=", expression];

type parameters = "<", type parameter list, ">";

type parameter list = type parameter, {",", type parameter};

type parameter = identifier;

constant definition = "constant", identifier, [":", type name], "=", expression;

function definition = "function", identifier, [type parameters], parameters, [":",
                      type name], "=", expression
                    | "private", "function", identifier, [type parameters], parameters, [":",
                      type name], "=", expression, function access control list
                    | "function", "(", overloadable operator, ")", parameters, [":",
                      type name], "=", expression
                    | "private", "function", "(", overloadable operator, ")", parameters,
                      [":", type name], "=", expression, function access control list;

function access control list = "access", function access control list block;

function access control list block = "{", {function access control entry}, "}";

function access control entry = signature;

group definition = "group", identifier, [type parameters], group block;

group block = "{", {group block item}, "}";

group block item = signature;

type definition = "type", identifier, [type parameters], "=", type name
                | "type", identifier, [type parameters], "=", "abstract", type name,
                  type access control list;

type access control list = "access", type access control list block;

type access control list block = "{", {type access control entry}, "}";

type access control entry = signature;
```

3.3 Declarations

```
declaration = initialized declaration
            | uninitialized declaration;

initialized declaration = uninitialized declaration, "=", expression;

uninitialized declaration = identifier list, ":", type name;

identifier list = identifier, {",", identifier};
```

```
type name = "(", type name, ")"
          | "infer"
          | "subtype", type name
          | basic type name
          | composite type name
          | function type name
          | identifier, [type arguments]
          | intersection type name
          | literal type name
          | medium address type name
          | medium type name
          | union type name;

basic type name = "bool" | "byte" | "char" | "float" | "int" | "string";

composite type name = array type name
                    | associative array type name
                    | product type name
                    | record type name
                    | sum type name
                    | variant type name;

array type name = "[", [expression], "]", type name;

associative array type name = "[", type name, "]", type name;

product type name = "product", product type block;

product type block = "{", {product type block item}, "}";

product type block item = include directive
                        | type name;

record type name = "record", record type block;

record type block = "{", {record type block item}, "}";

record type block item = include directive
                       | uninitialized declaration;

sum type name = "sum", sum type block;

sum type block = "{", {sum type block item}, "}";

sum type block item = include directive
                    | type name;

variant type name = "variant", variant type block;

variant type block = "{", {variant type block item}, "}";

variant type block item = include directive
                        | uninitialized declaration;

function type name = "function", "(", [type name list], ")", ":", type name;

type name list = type name, {",", type name};

intersection type name = "intersection", intersection type block;

intersection type block = "{", {intersection type block item}, "}";

intersection type block item = include directive
                             | type name;
```

```
literal type name = "literal", identifier;

medium address type name = buffer address type name
                         | channel address type name;

buffer address type name = "&", [interface access right], "buffer", type name;

channel address type name = "&", [interface access right], "channel", type name;

medium type name = buffer type name
                 | channel type name;

buffer type name = [interface access attribute], "buffer", ["<", expression, ">"], type name
                 | "private", "buffer", ["<", expression, ">"], type name,
                   [interface access control list];

channel type name = [interface access attribute], "channel", type name
                  | "private", "channel", type name, [interface access control list];

union type name = "union", union type block;

union type block = "{", {union type block item}, "}";

union type block item = include directive
                      | type name;
```

3.4 Expressions

```
expression = buffer extraction expression
           | buffer insertion expression;

buffer extraction expression = buffer extraction pattern, ">>", l-value expression;

buffer extraction pattern = l-value expression;

l-value expression = "(" l-value expression ")"
                   | "*", unary expression
                   | identifier
                   | postfix expression, ".", field
                   | postfix expression, "[", expression, "]";

unary expression = postfix expression
                 | "&", unary expression
                 | "+", unary expression
                 | "++", unary expression
                 | "-", unary expression
                 | "--", unary expression
                 | "*", unary expression
                 | "~", unary expression;

postfix expression = primary expression
                   | postfix expression, ".", field
                   | postfix expression, arguments
                   | postfix expression, indices;

primary expression = "(", expression, ")"
                   | "(", overloadable operator, ")"
                   | "self"
                   | "time"
                   | anonymous function expression
                   | compound expression
```

```
                    | identifier
                    | instantiation expression
                    | literal expression
                    | statement expression;

anonymous function expression = "function", parameters, [":", type name], "=",
                                compound expression;

compound expression = "{", [compound expression item list], "}";

compound expression item list = compound expression item, {[";"], compound expression item};

compound expression item = declaration
                         | expression
                         | labelled expression;

labelled expression = "case", expression, ":" expression
                    | "default", ":", expression;

instantiation expression = "create", anonymous actor, arguments
                         | "create", identifier, [type arguments], arguments
                         | "make", identifier, [type arguments]
                         | "make", medium type name
                         | "new", composite type name
                         | "new", identifier, [type arguments];

anonymous actor = "actor", actor block;

arguments = "(", [argument list], ")";

argument list = argument, {",", argument};

argument = expression;

literal expression = basic literal expression
                   | composite literal expression;

basic literal expression = literal;

composite literal expression = array literal expression
                             | associative array literal expression
                             | product literal expression
                             | record literal expression
                             | sum literal expression
                             | variant literal expression;

array literal expression = "[", [expression list], "]"
                         | "[", expression, generators, "]";

expression list = expression, {",", expression};

generators = generator, {generator};

generator = "for", b-value expression, "in", expression, {"if", expression};

b-value expression = identifier;

associative array literal expression = "[", [associative array pair list], "]";

associative array pair list = associative array pair, {",", associative array pair};

associative array pair = identifier, ":", expression;

product literal expression = "(", [expression list], ")";
```

```
record literal expression = "(", [record pair list], ")";

record pair list = record pair, {",", record pair};

record pair = identifier, ":", expression;

sum literal expression = "<", decimal literal, ":", expression ">";

variant literal expression = "<", identifier, ":", expression, ">";

statement expression = do-whenever expression
                     | do-while expression
                     | for expression
                     | if expression
                     | interrupt expression
                     | jump expression
                     | match expression
                     | parallel expression
                     | parallel for expression
                     | skip expression
                     | switch expression
                     | wait expression
                     | whenever expression
                     | while expression;

do-whenever expression = "do", expression, "whenever", expression, "in", interval expression;

interval expression = "(", expression, ",", "...", ")"
                    | "(", expression, ",", expression, ")"
                    | "(", expression, ",", expression, "]"
                    | "[", expression, ",", expression, ")"
                    | "[", expression, ",", expression, "]";

do-while expression = "do", expression, "while", expression;

for expression = "for", b-value expression, "in", expression, expression;

if expression = "if", expression, expression, ["else", expression];

interrupt expression = "interrupt", expression, interrupt handlers;

interrupt handlers = interrupt handler, {interrupt handler};

interrupt handler = "handle", expression, ["in", interval expression], expression;

jump expression = "abort"
                | "break"
                | "continue"
                | "exit"
                | "resume"
                | "return", [expression];

match expression = "match", expression, expression;

parallel expression = "parallel", expression, "and", expression;

parallel for expression = "parallel", for expression;

skip expression = "skip";

switch expression = "switch", expression, expression;

wait expression = "wait", expression, "in", interval expression, expression, ["else",
                  expression];
```

```
whenever expression = "whenever", expression, "in", interval expression, expression;

while expression = "while", expression, expression;

field = decimal literal
      | identifier;

indices = "[", index list, "]";

index list = index, {",", index};

index = expression;

buffer insertion expression = assignment expression
                            | buffer insertion pattern, "<<", expression;

assignment expression = parallel logical OR expression
                      | l-value expression, assignment operator, expression;

parallel logical OR expression = parallel logical AND expression
                               | parallel logical OR expression, "|||",
                                 parallel logical AND expression;

parallel logical AND expression = short-circuit logical OR expression
                                | parallel logical AND expression, "&&&",
                                  short-circuit logical OR expression;

short-circuit logical OR expression = short-circuit logical AND expression
                                    | short-circuit logical OR expression, "||",
                                      short-circuit logical AND expression;

short-circuit logical AND expression = inclusive OR expression
                                     | short-circuit logical AND expression, "&&",
                                       inclusive OR expression;

inclusive OR expression = exclusive OR expression
                        | inclusive OR expression, "|", exclusive OR expression;

exclusive OR expression = AND expression
                        | exclusive OR expression, "^", AND expression;

AND expression = equality expression
               | AND expression, "&", equality expression;

equality expression = inequality expression
                    | equality expression, "==", inequality expression;

inequality expression = relational expression
                      | inequality expression, "!=", relational expression;

relational expression = IN expression
                      | relational expression, "<", IN expression
                      | relational expression, "<=", IN expression
                      | relational expression, ">", IN expression
                      | relational expression, ">=", IN expression;

IN expression = IS expression
              | IN expression, "in", IS expression;

IS expression = event expression
              | event expression, "is", type name;

event expression = shift expression
                 | l-value expression, "@", shift expression;
```

```
shift expression = additive expression
                 | shift expression, "<<<", additive expression
                 | shift expression, ">>>", additive expression;

additive expression = multiplicative expression
                    | additive expression, "+", multiplicative expression
                    | additive expression, "-", multiplicative expression;

multiplicative expression = negation expression
                          | multiplicative expression, "%", negation expression
                          | multiplicative expression, "*", negation expression
                          | multiplicative expression, "/", negation expression;

negation expression = channel status expression
                    | "!", negation expression;

channel status expression = unary expression
                          | l-value expression, "?";

assignment operator = "%="   | "&&="  | "&="   | "*="   | "+="   | "-="   | "/="   | "^="
                    | "|="   | "||=";

buffer insertion pattern = l-value expression;
```

References

[1] John C. Eidson, Edward A. Lee, Slobodan Matic, Sanjit A. Seshia, and Jia Zou. Distributed real-time software for cyber-physical systems. *Proceedings of the IEEE*, 100(1):45–59, January 2012.

[2] International Organization for Standardization, Geneva, Switzerland. *ISO/IEC 14977:1996(E): Information technology – Syntactic metalanguage – Extended BNF*, 1996.

[3] Eleftherios Matsikoudis, Christos Stergiou, and Edward A. Lee. On the schedulability of real-time discrete-event systems. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–15, 2013.

[4] Yang Zhao, Jie Liu, and Edward A. Lee. A programming model for time-synchronized distributed real-time systems. In *Real Time and Embedded Technology and Applications Symposium, 2007. RTAS '07. 13th IEEE*, pages 259–268, April 2007.