The Vole Actor Language

Abstract

The Vole Actor language was created as part of a class project to help explore the possibilities of a virtual machine designed specifically to support the Actor Model of programming. Vole is an object-oriented Actor language that has shaped and has been shaped by the Actor VM. This paper describes Vole, the Vole compiler, and language-related aspects of the Actor VM.

Introduction

The Actor Model of programming [1] encapsulates state within independent, asynchronous active entities called Actors. Actors communicate by passing messages. An Actor's internal state can only be accessed by a message processed by that Actor.

Vole, an Actor Model programming language, was developed to help guide the creation of the Actor Virtual Machine (VM), and in turn was influenced by the VM as the two evolved. The Actor VM [8] manages each actor instance in an independent simulated thread of execution and provides a thread-safe messaging mechanism between actor instances. The register-based VM execute 64-bit word instructions that are loaded when the VM is started. The binary files can be created using the Actor VM assembler or the Vole compiler.

After describing some of the features of the Vole programming language, this paper will examine the Vole compiler and some aspects of the Actor VM. Then some ideas for future development will be discussed.

Vole

A number of Actor programming languages exist today [2], but few are *purely* Actor languages[1]. Vole is a pure Actor language because it provides no mechanism to circumvent the Actor model. The design of Vole leverages the similarity between Actors and active classes [4] to create an object-oriented language where Actors and classes are equivalent. The design of Vole was inspired by a number of computer languages, including (but not limited to): Ada, Smalltalk, Python, and Java.

[1] For instance, Scala and Erlang are Actor languages, but also functional languages. ActorFoundry is a Java Actor library so all non-Actor aspects of Java are still available. E does not encapsulate everything in Actors.

Message passing is a primary capability of Vole, and therefore must be efficient. Messages are the only direct communications allowed between threads of execution. Therefore, a limited number of primitive (and immutable) data types are included to facilitate thread-safe communication between Actors[2].

Vole uses a "curly brace" syntax similar to C++ and Java. A conscious decision was made to avoid obscure glyphs and instead to favor English keywords. Semicolons are used to delineate lines of code, messages, and even actors.

```
actor HelloWorld {
   constructor () {
      emit "Hello World!";
   };
};
```

Glyphs

The intent is to create a clear, clean, and minimal language that is easy to read by a majority of software engineers. To this end, an attempt is being made to minimize the use of symbols, especially where they could easily be misunderstood. However, many symbols are expected and traditional. It would be confusing if double quotes were not used to delineate strings, or if "+" was not used to denote addition. The tradition of semicolons, parentheses, brackets, and braces is continued.

Glyph	Usage
:	Specifies the type of an identifier in the definition of class attributes, message parameters, and local variables.
.	Separates components of a hierarchical name.
,	Separates elements of a list.
;	Separates stand-alone elements of a sequence, i.e. statements within a code block, messages within a class, attributes within a class, etc.
#	Specifies the start of a line comment.
( )	Specifies an argument definition or argument values, or a sub-expression.
[ ]	Specifies a tuple definition or selection of a tuple element.
{ }	Specifies a block of class elements (attributes and/or messages) or a block of code.
+	Addition operator.
-	Subtraction operator.

[2] Smalltalk uses classes for types like a basic integer. Primitive types have a relatively deep inheritance structure and long lists of methods. Vole tries to support operations in the VM as quickly as possible by directly manipulating primitive data types.

*	Multiplication operator.
/	Division operator.
**	Exponentiation operator.
<<	Bit shift left (up) operator.
>>	Bit shift right (down) operator.
%	Modulo operator; alternate form of “mod”.
&	And operator; alternate form of “and”.
|	Or operator; alternate form of “or”.
!	Not operator; alternate form of “not”.
:=	Assignment operator.
=	Equal operator.
==	Alternate form of equal operator.
>	Greater than operator.
>=	Greater than or equal operator.
<	Less than operator.
<=	Less than or equal operator.
/=	Not equal operator.
!=	Alternate form of not equal operator.
" "	String literal delimiter.
@	Decorator operator (proposed).
‘	Reflection operator (proposed).
#@	Tagged comment (proposed).

Actors
Messages
Attributes/Variables
Control Structures
Templates (no need for)
Tagged comments al a JavaDoc
Compiling
Aspects

Synchronous and asynchronous message passing is augmented with tail-call semantics to allow efficient recursion.

Vole Compiler

Since the Actor VM is written in Ada, open source Ada lex and yacc tools (aflex and ayacc [9]) were used to develop the Vole compiler. Source code can be freely downloaded from the University of California and is available in the Vole distribution.

Aflex and ayacc produce Ada 83 source code in files that do not have GNAT-compliant names, but a simple shell script is used to fix this. The same shell script also automatically removes the `constant` keyword from the declaration of `DEBUG` in "`kv-avm-vole_parser.adb`". This is the only post-generation modification of code from the parser tools.

Aflex is run against "`vole_lex.l`" and ayacc against "`vole.y`". A subset of the parser grammar is shown below in Extended Backus–Naur Form although the form used by ayacc is different.

The parser produced by ayacc builds an abstract syntax tree (AST) as it parsed source code. The compiler (main program "`volec.adb`") then uses a graph rewriter implemented using the Visitor Pattern [DP] to fill in implied nodes and to prepare connections. An additional visitor is used to create a DOT [?] output file with a graphic representation of the AST. Finally, a code generator visitor is used to emit AVM wordcode into a binary file.

Actor VM

Type Model

Similar to Java, the Actor VM divides the world into primitive types (of which there are 12) and actors. Objects (actor instances) exist within the virtual machine, but are not directly accessible. Instead, actor references (a primitive type) are used to "point" to an instance of an actor. Actor references can be passed in messages. Other primitive types include 64-bit signed integers, 64-bit floating-point numbers, 64-bit unsigned integers, immutable strings, immutable tuples, tuple maps (folding instructions), Booleans, Actor definitions (Actor types), message definitions, futures, and a null-object type called "Unset". Strings and tuples can't be modified, only replaced.

Future Development

The Vole language and compiler are so new that future development seems limitless. It is perhaps helpful to divide the work into two separate camps: recommended and speculative.

Recommended

Many standard language features need to be honed or filled out. For example, type checking needs to be added where possible. Module imports need to be implemented. The compiler does not yet support interface or class inheritance.

It should be possible to eliminate the need to declare local variables. Even if their type cannot be determined at compile time, their scope is clear and the VM supports dynamic typing.

As of yet, the distinction between message (externally invocable) and method (only available from within the class hierarchy) is not implemented.

One goal is to have a clean, understandable, and minimal language. To this end, syntactic sugar is useful for replacing boilerplate structures with a clean syntax.

An important feature to add as soon as possible is a Unit Test facility so that a unit's test code can exist with the unit and be automatically run at compile time. No separate framework should be needed to test code; it should all be part and parcel of the nominal tool chain.

That same nominal tool chain should render superfluous a separate `make` utility. Compiling a Vole program should be as easy as running `gnatmake` to build an Ada program.

Finally, no language is complete without a standard library to add capabilities that are not inherent in the language.

Speculative

This is the interesting camp. Design By Contact, Aspect weaving, reflection, migratable code, tuple spaces or blackboards, easy mixins using redirection of messages.

```
EBNF

program = _imports, actors;
```

```
_imports = (* unimplemented *)

actors = actor, “;”, _actors;
_actors = empty | actors;
actor = “actor”, identifier, “{“, _attributes, methods, “}”;
(* subclasses and interface implementation not currently implemented *)

_attributes = empty | attributes;
attributes = attribute, _attributes;
attribute = (“attribute”, identifier, “:”, type_init, “;”)
          | (“predicate”, identifier, “;”);
type_init = type [“:=”, expression];
type = “Integer” | “Unsigned” | “Actor” | “Boolean”; (* not all imp. *)

methods = method, “;”, _methods;
_methods = empty | method;
method = constructor | method | message | when_message;

constructor = “constructor”, arg_list, code_block;
method = “method”, identifier, arg_list, “returns”, arg_list,
code_block; (* methods are private messages *)
message = “message”, identifier, arg_list, “returns”, arg_list,
code_block;
when_message = “when”, identifier, message;

arg_list = “(“, _arg_defs, “)”;
_arg_defs = empty | arg_defs;
arg_defs = arg_def | arg_def “,” arg_defs;
arg_def = identifier, “:”, type;

code_block = “{“, _statements, “}”;
_statements = empty | statements;
statements = statement, “;”, _statements;
statement = vardef | assign | assert | emit | send | if | loop |
return;

vardef = “local”, identifier, “:”, type_init;
assign = identifier, “:=”, expression;
assert = “assert”, expression;
emit = “emit”, expression;
send = “send”, (send_gosub | send_actor);
send_gosub = “self”, “.”, identifier, arg_values;
send_actor = identifier, “.”, identifier, arg_values;
if = “if”, expression, “then”, “{“, _statements, rest_of_if;
rest_of_if = “}”, [ (“else”, code_block)
    | (“elseif”, expression, “then”, “{“, _statements, rest_of_if)];

loop = [“while”, expression], “loop”, code_block;

return = “return”, (empty | expression | tuple | call_gosub |
call_actor);

arg_values = “(“, _expression_list, “)”;

tuple = “[“, _expression_list, “]”;

_expression_list = empty | expression_list;
```

```
expression_list = expression, [“,”, expression_list];

call_gosub = “self”, “.”, identifier, arg_values;

call_actor = variable, “.”, identifier, arg_values;

variable = [“self”, “.”], identifier;

arg_values =

expression =
```

[1]......Agha, Gul Abdulnabi. "Actors: a model of concurrent computation in distributed systems." (1985).
[2] Karmani, Rajesh K., and Gul Agha. "Actors." Encyclopedia of Parallel Computing. Springer US, 2011. 1-11.
[3] Karmani, Rajesh K., Amin Shali, and Gul Agha. "Actor frameworks for the JVM platform: a comparative analysis." Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. ACM, 2009.
[4] Fowler, Martin. UML Distilled: A Brief Guide to the Standard Object Modeling Languange. Addison-Wesley Professional, 2004.
[5] Meijer, Erik, and John Gough. "Technical overview of the common language runtime." language 29 (2001): 7.
[6] TMS 9900 Microprocessor Data Manual, Texas Instruments, 1976
[7] website: http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf
[8] Kristola, David J. “Actor Virtual Machine” Unpublished, 2013.
[9] http://www.ics.uci.edu/~arcadia/Aflex-Ayacc/aflex-ayacc.html