Proceedings of the 11th Colloquium for Information Systems Security Education
Boston University
Boston, MA June 4-7, 2007

Sequential and Parallel/Concurrent Actor-Oriented Solutions of the Dominator Problem

Vojislav Stojkovic, William Lupton, *Morgan State University*

Abstract – The paper presents a known sequential and a new parallel/concurrent actor-oriented solution of the Dominator problem. The new parallel/concurrent actor-oriented Dominator algorithm computes sets of dominators of nodes of a given control flow graph in a parallel/concurrent actor-oriented way. The new Dominator algorithm is implemented as the multi-actor system in the Easel programming language. The new Dominator algorithm and its implementation are important contributions to the theory and practice of parallel / concurrent algorithms and actor-oriented programming. Because Dominator algorithm has applications in Information Assurance and Computer Security in detecting and locating program attacks – this novel and innovative Dominator algorithm may greatly influence these disciplines.

Index terms – Actors, Easel programming language, Dominator problem, control flow graph.

I. Background, Idea, and Contribution

Most investigations of the Dominator problem have focused on sequential, iterative, and efficient computations of sets of dominators of nodes of a given control flow graph. Our research focuses on parallel / concurrent actor-oriented computation of sets of dominators of nodes of a given control flow graph. The parallel/concurrent actor-oriented approach improves clarity and efficiency of the solution of the Dominator problem.

The main idea of this paper/work is to solve the Dominator problem using the parallel/concurrent actor-oriented approach. For that purpose a sequential Dominator algorithm has been transformed into a new and unique parallel/concurrent actor-oriented Dominator algorithm. This new Dominator algorithm and its implementation are important contributions to the theory and practice of parallel/concurrent algorithms and actor-oriented programming. Because Dominator algorithms have numerous applications in Information Assurance and Computer Security in detecting and locating program vulnerabilities and attacks – the new Dominator algorithm may have strong influence on these critical areas.

To make our work more tangible we elected to use the Easel actor-oriented programming language as the implementation language.

II. Easel

Easel – An **E**mergent **A**lgorithm **S**imulation **E**nvironment and **L**anguage - is a new general-purpose, actor-oriented, modeling, simulation, visualization, and animation programming language. [1, 2, 3, 4]. Easel was designed to work with large numbers of locally interacting actors – autonomous participants of a system/process. Easel was intended as a research software tool for handling unbounded systems, emergent algorithms and survivability architectures. Easel can assist users in predicting behavior and interactions of components / elements and processes/systems, if they are implemented – programmed in Easel. [7, 8, 9]. Easel can be used for describing, modeling, simulating, visualizing, and animating complex systems/processes with dynamic attributes such as: large numbers of interacting components and/or participants, lack of central control and incomplete and imprecise information. These systems / processes are characterized by local neighbor interactions, limited visibility or knowledge between internal components-elements and larger surrounding systems/processes, and system-wide emergent properties.

In Easel one of the key types is the built-in actor type. The actor type has the special property of being threaded – that is to be born, to live, and to die. Basically - actors are primitive agents.

III. Actors

Actors are self-contained, autonomous, independent, interacting computing elements, which encapsulate behaviors (data and procedure and/or function) as well as processes. Each actor performs its tasks asynchronously and communicates with each other by sending and receiving messages. Actors may be dynamically created and reconfigured, which provides considerable flexibility in organizing concurrent activity.

Morgan State University
School of Computer, Mathematical, and Natural Sciences
Computer Science Department
Baltimore, MD 21251

For the life of the process, the actor:
- has its own memory
- requires (some) CPU time:
  - to update its internal state and
  - to interact with other actors.

Each actor may have both global and local properties. The global properties of actor are defined in the actor parameter list. The local properties of actor are defined in the actor body.

On a single-processor computer, actors are executed concurrently or in other words occurring and/or operating in the same time interval.

On a multi-processor computer, actors are executed in parallel or in other words occurring and/or operating at the same time.

Two advantages to using actors for building multi-agent systems include:
- actors provide a logically distributed programming model which allow systems to be decomposed into autonomous, interacting components without the need to explicitly manage concurrency
- the performance gains that may be achieved by running actor implementations on parallel and distributed architectures will make it possible to find solutions to large and complex real world problems.

Actors provide a natural extension of the object-oriented paradigm to the agent-oriented paradigm, which is concurrent and distributed computation. [5, 6].

Actors are natural extensions of classes. To evolve from object-oriented programming to actor-oriented programming the following three steps are necessary:
- develop actor-oriented data structures
- develop/design algorithms based on actor-oriented data and control structures and actors, and
- code algorithms - write programs using an actor-oriented programming language.

IV. Control Flow Graph

An edge is a pair of nodes. An edge may be an undirected edge or a directed edge. An edge is an undirected edge, if it connects two nodes in both directions. An edge is a directed edge, if it connects two nodes in one direction.
$X \rightarrow Y$ is a direct edge that connects the node X and the node Y.
If $X \rightarrow Y$ then the node X is a predecessor of the node Y and the node Y is a successor of the node X.
PRED(X) is the set of all predecessors of the node X.
$PRED(X) = \{Y \mid Y \rightarrow X\}$.
SUCC(X) is the set of all successors of the node X.
$SUCC(X) = \{Y \mid X \rightarrow Y\}$.

Definition-1

A graph is the duple <V, E> where:
- V is a set of nodes (or vertices);
- E is a set of edges, where E is a subset of $V \times V$.

Definition-2

A directed graph is a graph in which all edges are directed edges.

$<X_0, X_1, \ldots, X_{k-1}, X_k>$ is a path from the node X_0 to the node X_k ($k \geq 0$) such that $X_0 \rightarrow X_1, \ldots, X_{k-1} \rightarrow X_k$.
$X \rightarrow^* Y$ is a path from the node X to the node Y.

The node Y is reachable from the node X if there is a path from the node X to the node Y.

Direct graphs may be used to represent various data and control structures. [10].

Definition-3

A control flow graph (CFG) is the triple <V, E, Entry> where:
- <V, E> is a directed graph
- Entry ε V is a unique node and
- all nodes in a control flow graph are reachable from the node Entry.

Control flow graphs are the most important direct graphs. One characteristic of control flow graphs that distinguishes them from other direct graphs is the presence of a unique node Entry from which all other nodes in the control flow graph are reachable.

Example-1

```
<
 {a, b, c, d, e, f, g, h, i, j},
 {
  (a, b), (b, c), (c, d), (c, i), (d, e), (d, f), (e, g), (f, g),
  (g, d), (g, h), (h, c), (i, j)
 },
 a
>
```

is an example of a control flow graph.

All nodes are reachable from the entry node a.
$a \rightarrow^* a$ because there is the path <a>
$a \rightarrow^* b$ because there is the path <a, b>
$a \rightarrow^* c$ because there is the path <a, b, c>
$a \rightarrow^* d$ because there is the path <a, b, c, d>
$a \rightarrow^* e$ because there is the path <a, b, c, d, e>
$a \rightarrow^* f$ because there is the path <a, b, c, d, f>

a→*g because there is:
 the path <a, b, c, d, e, g> or
 the path <a, b, c, d, f, g>
a→*h because there is
 the path <a, b, c, d, e, g, h> or
 the path <a, b, c, d, f, g, h>
a→*i because there is the path <a, b, c, i>
a→*j because there is the path <a, b, c, i, j>.

The given control flow graph can be represented graphically as shown in Figure 1.

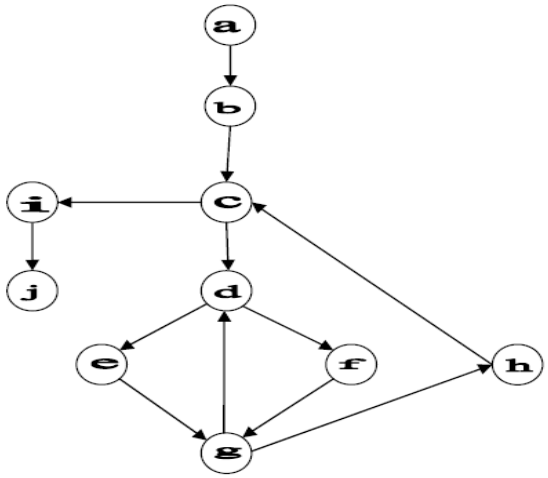

FIGURE 1.

The given control flow graph can be represented in the Easel programming language as a list:

```
cfg:: list :=
 (list)
 '[
    (list)'["a", "b", "c", "d", "e", "f", "g", "h", "i", "j"],
    (list)
    '[
       (list)'["a", "b"],
       (list)'["b", "c"],
       (list)'["c", "d"],
       (list)'["c", "i"],
       (list)'["d", "e"],
       (list)'["d", "f"],
       (list)'["e", "g"],
       (list)'["f", "g"],
       (list)'["g", "d"],
       (list)'["g", "h"],
       (list)'["h", "c"],
       (list)'["i", "j"]
    ],
    "a"
];
```

The set of nodes (or vertices) V can be extracted from the list cfg in the following way:

```
V:: List := First(cfg);
```

where

```
First(L: List): any is
 return car(L);
```

The set of edges E can be extracted from the list cfg in the following way:

```
E:: List := Second(cfg);
```

where

```
Second(L: List): any is
 return car(cdr(L));
```

The node Entry can be extracted from the list cfg in the following way:

```
Entry:: string := Third(cfg);
```

where

```
Third(L: List): any is
 return car(cdr(cdr(L)));
```

The type List is defined in the following way:

```
List: type is
 list any;
```

V. EASEL IMPLEMENTATION OF A CFG

A control flow graph can be implemented in the Easel programming language as a multi-actor system. Each node of the given control flow graph is implemented as an actor.

An environment-world of a control flow graph multi-actor system can be implemented in the Easel programming language as the simulation type CFG:

```
CFG: simulation type is
 nodes:: list := new list any;
```

CFG simulation type is used as a common zone for sharing the data - in our case the actors' properties.
In other words - the set of nodes (or vertices) V is implemented as the list of nodes, where each node can be implemented as the appropriate/specific/different actor.

```
node(n: string, V: list, E: list, Entry: string): actor type is
# Properties
 dominators:: List := ?;
 old_dominators:: List := ?;
 predecessors:: List := ?;
 s:: List := ?;
 inter:: List := ?;
 changed:: boolean := ?;
# ? – question mark symbol means that
# the initial-starting value is not assigned.
# Initialization
 if n == Entry
 then
  dominators := makeSet(Entry);
 else
  dominators := V;
  changed := true;
  predecessors := pred(n, E);
  if Null(predecessors)
  then
   inter := (list)'[];
  else
   inter := V;
# Synchronization
  for every 1..100 do wait 10.0;
# Life cycle
 outln("DominatorSet(", n, ") = ", dominators);
```

The global properties of the node actor are:
- the name of node n
- the set of nodes (or vertices) V
- the set of edges E
- the node Entry

The local properties of the node actor are:
- the list of nodes dominators
- the auxiliary list of nodes old_dominators
- the list of nodes predecessors
- the auxiliary list of nodes s
- the auxiliary list of nodes inter
- the flag changed

The life cycle of the node actor will be explained later.

The function makeSet from the given element makes the single element set – list.

```
makeSet(e: any): List is return cons(e, (list)'[]);
```

The function pred from the given node and the list of edges compute the list of predecessors of the given node.

```
pred(n: string, E: list): List is
 if (Null E)
 then
  return (list)'[];
 else
  if (Second(First E)) == n
  then
   return cons(First(First(E)), pred(n, cdr(E)));
  else
   return pred(n, cdr(E));
```

VI. Set of Dominators

The following explanation assumes that nodes (vertices) and edges belong to a control flow graph.

Definition-4

A node X dominates a node Y, written X DOM Y, if every path from the Entry node to the node Y includes the node X.

By the Definition-4:
- every node dominates itself,
for all X: X DOM X
- the Entry node dominates every node X, including itself,
Entry DOM X, Entry DOM Entry.

If X DOM Y, then any path p: Entry →* Y can be split into two parts: p-prefix: Entry →* X and p-suffix: X →* Y.

Definition-5

The dominator set of the node Y, written DOM(Y), is the set of nodes Xs that dominate the node Y, X DOM Y.

DOM(Y) = {X | X DOM Y}.

The DOM relation can be represented by the dominator set DOM(X). DOM(X) ≠ { }, since X is in DOM(X).

Example-2

The dominator set for each node of the given CFG is shown in Table 1 below.

X	**DOM(X)**
A	{a}
B	{a, b}
C	{a, b, c}
D	{a, b, c, d}
E	{a, b, c, d, e}
F	{a, b, c, d, f}
G	{a, b, c, d, g}
H	{a, b, c, d, g, h}
I	{a, b, c, i}
J	{a, b, c, i, j}

TABLE 1.

ISBN 1-933510-96-7/$15.00 ©2007 CISSE

Definition-6

A node X strictly dominates a node Y, written X SDOM Y, if X DOM Y and X ≠ Y.

Definition-7

The strict dominator set of the node Y, SDOM(Y), is the set of nodes Xs that strictly dominate the node Y, X SDOM Y.

SDOM(Y) = {X | X SDOM Y} = DOM(Y) – {Y}.

By Definition-7:
- no node can strictly dominate itself
- the strict dominator set may be empty.

Definition-8

A node X is an immediate dominator of a node Y, written X IDOM Y, if the node X is the nearest strict dominator of the node Y.

X IDOM Y, iff
X SDOM Y and (All Z | Z SDOM Y)[Z DOM X]

VII. Dominator Algorithm

The set of dominators of the node Y, DOM(Y), may be calculated using the following relation:

DOM(Y) =
{Y} Union Intersection(DOM(X) | X ε predecessors(Y))

In other words, the dominator set of the node Y includes the node Y and any other node that also dominates all of its predecessors.

Dominator algorithm is based on the previous relation – recursive definition. Dominator algorithm always terminates, since the values assigned to DOM(Y) are always a subset of the values from the previous iteration of the loop. When Dominator algorithm terminates, the value of DOM(Y) is the set of nodes that dominate the node Y.

The set of dominators of each node of the given control flow graph can be computed in two different ways:
- without using actors
- using actors.

The first solution is the sequential/classical solution which determines the sequential Dominator algorithm. The second solution is a parallel/concurrent actor-oriented innovative solution that determines the parallel/concurrent actor-oriented Dominator algorithm.

VIII. Easel Implementation of the Sequential Dominator Algorithm

The sequential Dominator algorithm is not unique. The following Easel program is an original, Easel, non actor, list based implementation of the sequential Dominator algorithm.

The program computes the set of dominators of each node of a given control flow graph.

The core of the program is the function SeqOfDomsSets to compute the sequence of dominators sets.

```
SeqOfDomsSets(V: list, E: list, Entry: string): List is

 # Declarations

 Dom:: List := ?;
 EPDom:: List := ?;
 SOfDsSs:: List := (list)'[];
 VdEntry:: List := ?;
 changed:: boolean := ?;
 OldDom:: List := ?;
 predecessors:: List := ?;
 inter:: List := ?;
 s:: List := ?;
 indexOfe:: int := ?;
 indexOfp:: int := ?;

# Initialization

 for e: each V do
  indexOfe := indexof(V, e);
  predecessors := pred(e, E);
  if e == Entry
  then
   Dom := makeSet(Entry);
  else
   Dom := V;

 # Computation

 EPDom := makeEPDom(e, predecessors, Dom);
 insert(SOfDsSs, indexOfe, EPDom);

# Iteration

 changed := true;
 for every (changed = true) do
  changed := false;
  for x: each V do
   if x !== Entry
   then
    indexOfe := indexof(V, x);
    OldDom := Third(SOfDsSs[indexOfe]);
    predecessors := Second(SOfDsSs[indexOfe]);
```

```
   if Null(predecessors)
   then
    inter := (list)'[];
   else
    inter := V;

   # Intersection

   for p: every predecessors do
    indexOfp := indexof(V, p);
    s := Third(SOfDsSs[indexOfp]);
    inter := intersection(inter, s);

   # Union

   Dom := union(makeSet(x), inter);
   SOfDsSs[indexOfe][2]:=Dom;
   if EqualList(Dom, OldDom) = false
   then
    changed := true;

 return SOfDsSs;
```

The list SOfDsSs has the following structure:

[EpDom, EPDom, …, EPDom]

where EPDom is the list with the following structure:

[e, predecessors, Dom]

where:
- e is a string - the name of the node
- predecessors is the list of predecessor nodes of the given node e
- Dom is the list of dominators of the given node e.

The function outputSeqOfDomsSets outputs the sequence of dominators sets.

```
outputSeqOfDomsSets(SOfDsSs: List): action is
 for s: every SOfDsSs do
  outln("DominatorSet(", First(s), ") = ", Third(s));
```

The body of the program is:

```
List: type is list any;
S:: List;
ctf:: list := …
(list) ’[ (list) ’[ …], (list) ’[ …], “a” ];
V:: List := First(cfg);
E:: List := Second(cfg)
Entry:: string := Third(cfg);

S := SeqOfDomsSets(V, E, Entry);
outputSeqOfDomsSets(S);
```

IX. Easel Implementation of the Parallel/Concurrent Actor-Oriented dominator Algorithm

The parallel/concurrent actor-oriented Dominator algorithm is quite unique. The following Easel program is a novel Easel actor-list based implementation of the parallel/concurrent actor-oriented Dominator algorithm.

The program computes the set of dominators of each node of a given control flow graph.

The core of the program is the actor node to compute the set of dominators of the given node of the given control flow graph.

```
CFG: simulation type is
 nodes:: list := new list any;

node(n: string, V: list, E: list, Entry: string): actor type is

# Properties
# Life cycle - Iteration

 for every (changed = true) do
  changed := false;
  old_dominators := dominators;
  for x: every predecessors do
   # Intersection
   for ss: each sim.nodes do
    if ss.n == x
    then
     s := ss.dominators;
     inter := intersection(inter, s);
     # Union
     dominators := union(makeSet(n), inter);
     if EqualList(dominators, old_dominators) = false
     then
      changed := true;
   # Result Output
   outln(“DominatorSet(“, n, “) = ", dominators);
```

The procedure activate_nodes for each node of a given control flow graph:
- creates/activates the appropriate actor node and
- places the actor node in the CFG_Sim.nodes array.

```
activate_nodes(V: List, E: List, Entry: string): action is
 CFG_Sim:: CFG := new CFG;
 for i: each 0 .. (length V)-1 do
  push
  (
   CFG_Sim.nodes,
   new(CFG_Sim, node(V[i], V, E, Entry))
  );
```

The body of the program is:

```
List: type is list any;

ctf:: list := …
V:: List := First(cfg);
E:: List := Second(cfg)
Entry:: string := Third(cfg);

activate_nodes(V, E, Entry);
```

X. Auxiliary Procedures and Function

As in any program, there are some auxiliary procedures and functions.

```
# Checks the emptiness of the list.

Null(L: List): boolean is
 return (length L) = 0;

# Computes the head – the first element of the list.

car(L: List): any is
 if Null(L)
 then
  outln("car: EMPTY LIST");
 else
  return L[0];

# Computes the tail – the rest of the list.

cdr(L: List): List is
 if (Null L) = 0
 then
  outln("cdr: EMPTY LIST");
 else
  if (length L) = 1
  then
   return (list) '[];
  else
   return get(L, 1, (length L)-1);

# Creates the pair.

cons(e: any, L: List): List is
 return catenate(list '[e], L);

# Checks equality of two lists.

EqualList(L1: List, L2: List): boolean is
 if Null(L1)
 then
  return true;
 else
  if (car L1) !== (car L2)
  then
   return false;
  else
   return EqualList(cdr(L1), cdr(L2));

# Checks membership.

Member(x: any, L: List): boolean is
 if Null(L)
 then
  return false;
 else
  if x == car(L)
  then
   return true;
  else
   return Member(x, cdr(L));

# Computes the intersection of two sets – lists.

intersection(L1: List, L2: List): List is
 if Null(L1)
 then
  return L1;
 else
  if Null(L2)
  then
   return L2;
  else
   if Member(car(L1), L2)
   then
    return cons(car(L1), intersection(cdr(L1), L2));
   else
    return intersection(cdr(L1), L2);

# Computes the union of two lists.

union(L1: List, L2: List): List is
 if Null(L1)
 then
  return L2;
 else
  if Null(L2)
  then
   return L1;
  else
   if Member(car(L1), L2)
   then
    return union(cdr(L1), L2);
   else
    return cons(car(L1), union(cdr(L1), L2));

# Makes E-P-Dom list.

makeEPDom(E: any, P: List, Dom: List): List is
 return cons(E, cons (P, makeSet(Dom)));
```

```
# Computes the index of the element in the list.

indexof(L: List, e: any): int is
 if e == car(L)
 then
  return 0;
 else   return 1+ indexof(cdr(L), e);
```

XI. Result of the Computation

Both programs produce the same output as follows:

```
DominatorSet(a) = list'[ "a"]
DominatorSet(b) = list'[ "b", "a"]
DominatorSet(c) = list'[ "c", "a", "b"]
DominatorSet(d) = list'[ "d", "a", "b", "c"]
DominatorSet(e) = list'[ "e", "a", "b", "c", "d"]
DominatorSet(f) = list'[ "f", "a", "b", "c", "d"]
DominatorSet(g) = list'[ "g", "a", "b", "c", "d"]
DominatorSet(h) = list'[ "h", "a", "b", "c", "d", "g"]
DominatorSet(i) = list'[ "i", "a", "b", "c"]
DominatorSet(j) = list'[ "j", "a", "b", "c", "i"]
```

XII. Applications

Each program may be broken up into a set of basic blocks. A basic block is a sequence of zero or more statements with no branch statements, except, perhaps the last statement, and no branch targets or labels, except, perhaps the first statement. Each basic block may correspond to a straight-line sequence of statements or even a single statement.

In a control flow graph:
- nodes represent basic block
- edges represent flow of control between basic blocks.

A conditional branch is represented by a node with two or more graph successors.

The Dominator problem is quite applicable to such areas as information assurance and computer security - especially in detecting program attacks. For example viruses/worms can cause changes of the dominator sets. In other words, any change of a dominator set indicates / suggest that the program is being attacked. Therefore, in order to protect data and programs (control structures) it is critical/important to have tools to compute dominator sets and to detect changes in them.

XIII. Conclusion

This paper provides:
- an implementation of a familiar sequential Dominator algorithm in the Easel programming language
- an implementation of a new and innovative parallel / concurrent actor-oriented Dominator algorithm in the Easel programming language.

The development of a unique Dominator algorithm and program are important contributions to the theory and practice of parallel/concurrent algorithms and actor-oriented programming.

This actor-oriented avant-garde approach may yet make the current sequential object-oriented approach obsolete.

XIV. References

[1] A.M. Christie, “Network Survivability Analysis Using Easel”, Technical Report, CMU/SEI, Pittsburgh, PA, 2002.
[2] A.M. Christie, D. Durkee, D.A. Fisher, and D.A. Mundie, “Easel Language Reference Manual and Author’s Guide”, Technical Report, CMU/SEI, Pittsburgh, PA, 2003.
[3] M. DeSantis, “Using Easel to Study Complex Systems”, news@sei interactive, CMU/SEI, Pittsburgh, PA, 2001.
[4] D.A. Fisher, “Design and Implementation of EASEL – A Language for Simulating Highly Distributed Systems”, CMU/SEI, Pittsburgh, PA.
[5] W. Lupton and V. Stojkovic, “Solving Incomplete and Incorrect Information Problems Using Conditional Planning, Execution Monitoring, and Situated Planning Agents”, The Proceedings of 12th Annual ASEET Symposium, Monterey, CA, 1998.
[6] V. Stojkovic and W. Lupton, “Software Agents – A Contribution to Agents Specification”, ISECON 2000, Philadelphia, PA, 2000.
[7] V. Stojkovic, G. Steele and W. Lupton, “An Agent-Oriented Approach to Find All Solutions to the Longest Common Subsequence (LCS) Problem”, Advances in Computational Intelligence and Security, *The Workshop of 2005 International Conference on Computational Intelligence and Security Proceedings*, Xi’an, China, 2005.
[8] V. Stojkovic and H. Huo, “Software Agents Action Securities”, 2006 International Conference on Computational Intelligence and Security (cis'2006) Guangzhou, China, 2006.
[9] V. Stojkovic, G. Steele, and W. Lupton, “Using Easel for Modeling and Simulating the Interactions of Cells in Order to Better Understand the Basics of Biological Processes and to Predict Their Likely Behaviors”, *The Proceedings of the 2003 IEEE Bioinformatics Conference*, Stanford University, Stanford, CA, 2003.
[10] M. Wolfe, High Performance Compilers for Parallel Computing, Addison-Wesley Publishing Company, 1996.