Static Detection of Access Anomalies in Ada95

Bernd Burgstaller, University of Sydney

Johann Blieberger, Vienna University of Technology

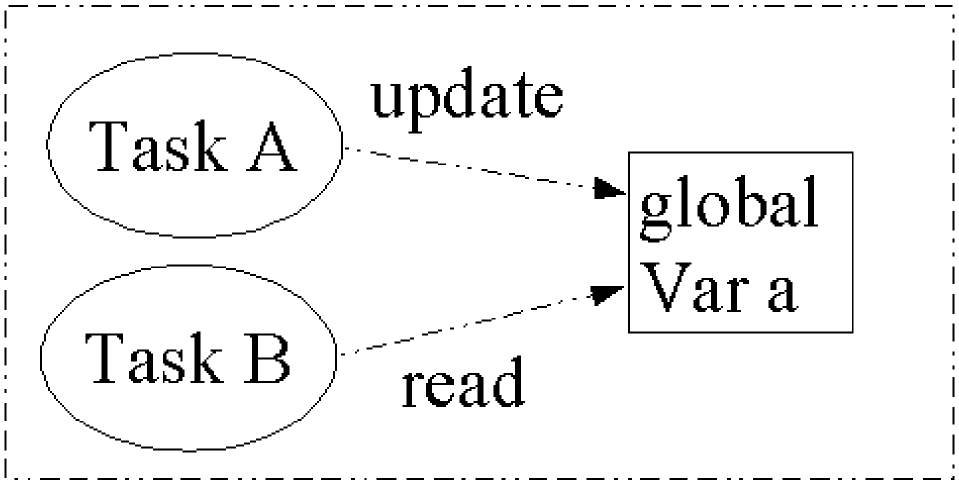
Robert Mittermayr, ARC Seibersdorf research GmbH

Outline

- The Problem / Goal
- Overview of our approach
- Framework for finding tasks running in parallel ($\|$-relation)
- Framework for determining sets of used and modified variables
- Conservative approach / Reducing false positives
- Complexity
- Summary / Outlook

The Problem / Goal

- Problem: Nondeterministic behavior of concurrent programs because of dynamic execution order of the statements ⇒ Access anomalies; also called
  - data races
  - non-sequential or unsynchronized accesses
- Goal: Find all access anomalies in Ada multi-tasking programs



Our Approach

- Static analysis (only the static structure of the program is taken into account)
- Two data flow frameworks for finding
  - tasks which potentially run in parallel ($\|$-relation)
  - sets of used and modified variables
- Conservative approach ($\Rightarrow$ false positives)
- Flow-insensitive ($\Rightarrow$ false positives); even if the intra-task structure of the program prevents parallel access our approach detects access anomalies.

$\|$-relation

- Given a CFG$(t) = (N, E, r)$ of a task body t, the basis for the data flow framework are standard equations of the form

$$\mathsf{S}_{\mathsf{out}}(n) = \mathsf{Gen}(n) \cup (\mathsf{S}_{\mathsf{in}}(n) \setminus \mathsf{Kill}(n))$$

$$\mathsf{S}_{\mathsf{in}}(n) = \bigcup_{n' \in \mathsf{Pred}(n)} \mathsf{S}_{\mathsf{out}}(n'),$$

  where n denotes a node of a CFG,
- Gen(n): set of task objects generated in node n. If an array of tasks is declared we model this by writing $t^* \in$ Gen(n).
- Kill(n): set of terminating task objects in node n.
- Since a compiler has to know the (cfg) nodes where a task is being generated or terminated we assume that these sets are available.

$\|$-relation(2)

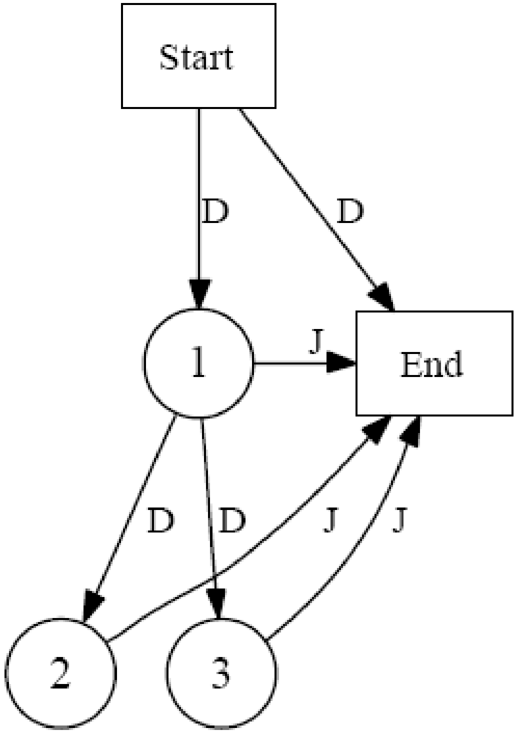
In order to determine the $\|$-relation from the solution of the data flow framework, we use the following algorithm.

CONSTRUCT$\|$ $()$

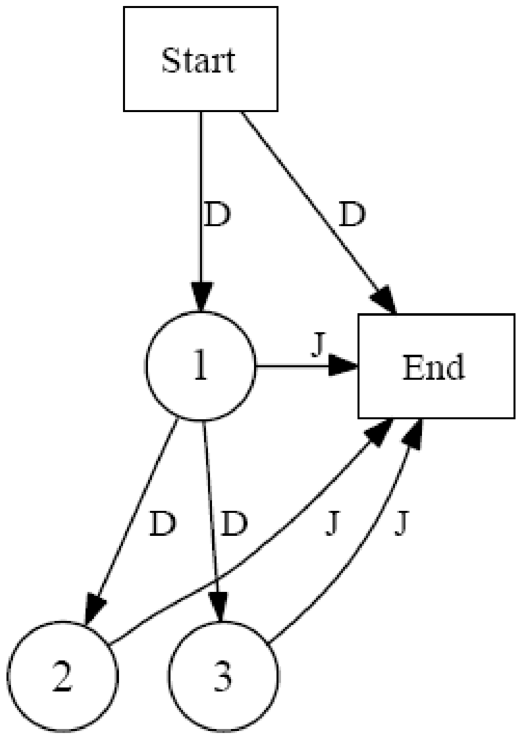
for *each task CFG* **do**
  for *each node* n **do**
    for *each* $t^* \in S(n)$ **do**
      DEFINE $t \parallel t$
    endfor
    for *each pair* $t_1, t_2 \in S(n)$ **do**
      DEFINE $t_1 \parallel t_2$
    endfor
  endfor
endfor

Example

```
procedure Main is
   task type task1 is          -- Node 1
   end task1;                  -- Node 1
   task type task2 is          -- Node 1
   end task2;                  -- Node 1
   task body task1 is begin
      -- do something          -- Node 2
   end task1;
   task body task2 is begin
      -- do something          -- Node 3
   end task2;
   t1 : task1;                 -- Node 1
   t2 : task2;                 -- Node 1
begin
   null;                       -- Node 1
end Main;
```



Example (2)



$$S(\text{Start}) = \{Main\},$$

$$S(1) = (S(\text{Start}) \backslash \text{Kill}(1)) \cup \text{Gen}(1)$$

$$= (\{Main\} \backslash \emptyset) \cup \{t1, t2\}$$

$$= \{Main, t1, t2\},$$

$$S(2) = S(1) = \{Main, t1, t2\},$$

$$S(3) = S(2) = \{Main, t1, t2\}.$$

After applying $CONSTRUCT \parallel$

$$Main \parallel t1$$

$$Main \parallel t2$$

$$t1 \parallel t2$$

Determining sets of used and modified variables

Unit u: a subprogram, task body, entry body, or dispatching operation u.

- u *owns* an entity e, if e is local to the declarative region of u.
- Task entries own the union of the entities owned by their corresponding accept statements.
- u owns all entities owned by entities called by u.

The ownership relation is reflexive and transitive.

Entities which are *visible* to an entity owned by u, but which are not owned by u, are said to be *global* to u.

We write $\mathcal{O}(u)$ to denote the set of entities owned by u, and $\mathcal{G}(u)$ to denote the set of entities that are global to u.

Determining sets of used and modified variables(2)

For every unit u that is a task body, and for the subprogram body corresponding to the environment task (the “main” program), our analysis determines

1. $\mathcal{O}_r$ and $\mathcal{O}_w$: sets of read/written variables owned by u,
2. $\mathcal{G}_r$ and $\mathcal{G}_w$: sets of read/written variables global to u, and
3. sets $\sigma_r = \mathcal{O}_r \cup \mathcal{G}_r$, $\sigma_w = \mathcal{O}_w \cup \mathcal{G}_w$, $\sigma_G = \mathcal{G}_r \cup \mathcal{G}_w$, and $\sigma_{rw} = \sigma_r \cup \sigma_w$.

We determine the quadruple $\langle \mathcal{O}_r, \mathcal{O}_w, \mathcal{G}_r, \mathcal{G}_w \rangle$ with small adaption to *“Interprocedural Side-Effect Analysis in Linear Time”* and *“Fast Interprocedural Alias Analysis”* by Cooper and Kennedy in 1988 and 1989 respectively.

Non-sequential access criterion

Predicate $\sigma(t_1, t_2)$ is *true* if some variable v is non-sequentially accessed by task objects t_1 and t_2 ($t_1 \parallel t_2$), *false* otherwise. It is formally defined as

$$\sigma(t_1,t_2) = \bigwedge_{v \in S} \Bigg[\bigg[\Big(\mathrm{use}(v,t_1) \wedge \mathrm{mod}(v,t_2)\Big) \qquad (1)$$

$$\vee \Big(\mathrm{mod}(v,t_1) \wedge \mathrm{use}(v,t_2)\Big) \qquad (2)$$

$$\vee \Big(\mathrm{mod}(v,t_1) \wedge \mathrm{mod}(v,t_2)\Big) \bigg] \qquad (3)$$

$$\wedge \Big(v \in \sigma_G(B(t_1)) \cup \sigma_G(B(t_2))\Big) \Bigg], \qquad (4)$$

where $S = \sigma_{rw}(B(t_1)) \cap \sigma_{rw}(B(t_2))$ are the variables accessed by both, $B(t_1)$ and $B(t_2)$, and (4) ensures that variable v is global to at least one of the involved task bodies.

Example

```
procedure Main is
    a : Integer := 0;
    task body task1 is begin
        for i in 1..10 loop
            a := i;
            --  do something else in the meantime
        end loop;
    end task1;
    task body task2 is begin
        for j in 1..10 loop
             --  read global variable a
        end loop;
    end task2;
    t1 : task1;
    t2 : task2;
begin
    ...
```

Example (2)

$$\mathcal{O}(t1) = \{i\}, \mathcal{O}(t2) = \{j\}, \mathcal{O}(Main) = \{a, i, j, t1, t2\}.$$
$$\mathcal{G}_w(t1) = \{a\}, \mathcal{G}_r(t1) = \emptyset,$$
$$\mathcal{G}_w(t2) = \emptyset, \mathcal{G}_r(t2) = \{a\},$$
$$\mathcal{G}_w(Main) = \mathcal{O}_w(Main) = \mathcal{G}_r(Main) = \mathcal{O}_r(Main) = \emptyset.$$

- $t1 \parallel t2$ and
- $\sigma_{rw}(B(t1)) \cap \sigma_{rw}(B(t2)) = \{a\}$ and
- $\sigma(t1, t2) =$ *true*
  $\Rightarrow$ access anomaly between $t1$ and $t2$ with respect to variable a.

Conservative approach

- Pointer (with respect to aliasing): every entity possibly targeted by a pointer is modified.
- Dispatching operations on tagged types: if the controlling tag can not be determined at compile-time $\Rightarrow$ assume procedure calls to all possible targets of the dispatching call.
- Coarse granularity of the $\|$-relation.

Reducing false positives

- We do not consider (as none of them can give raise to access anomalies)
  - variables marked by pragmas Atomic or Volatile
  - protected variables
  - modification that is due to an initialization expression of a declaration in the declarative_part
- Transitivity of owned relation

Complexity

- Computation of $\|$-relation:

$$O(|E| \cdot \log |N|)$$

where $|N|$ denotes the number of nodes and $|E|$ the number of edges in a CFG.

- Finding sets of used and modified variables:

$$O(|E| \cdot |N| + |N|^2)$$

with $|N|$ and $|E|$ being the number of call graph nodes and edges.

Summary / Outlook

- Our approach is able to handle most programs of practical importance
- Efficient
- Easy to implement
- Conservative $\Rightarrow$ false positives
- In the future we plan to apply symbolic analysis to this problem. Symbolic analysis is capable of incorporating flow-sensitive side-effects of a program. Thus reduces the number of false positives.

Thank you for your attention!

Questions?