Static Detection of Access Anomalies in Ada95*

Bernd Burgstaller[1], Johann Blieberger[2], and Robert Mittermayr[3]

[1] School of Information Technologies, The University of Sydney, Australia
bburg@it.usyd.edu.au
[2] Institute for Computer-Aided Automation, TU Vienna
Treitlstr. 1, A-1040 Vienna, Austria
blieb@auto.tuwien.ac.at
[3] ITS Softwaresysteme, ARC Seibersdorf research GmbH, TechGate Vienna
Donau-City-Str. 1, A-1220 Vienna, Austria
robert.mittermayr@arcs.ac.at

Abstract. In this paper we present data flow frameworks that are able to detect access anomalies in Ada multi-tasking programs. In particular, our approach finds all possible non-sequential accesses to shared non-protected variables. The algorithms employed are very efficient. Our approach is conservative and may find false positives.

1 Introduction

Concurrent programming is a complex task. One reason for this is that scheduling exponentially increases the possible program states. Thus a dynamic execution order of the statements executed in parallel is introduced. In general this leads to different behavior between different runs of a program, even on the same input. Because of the nondeterministic behavior, faults are difficult to detect. Static program analysis, which has been used since the beginning of software, can be a valuable aid for the detection of such faults.

One of the major problems with concurrent programming are *access anomalies*, also called *data races*. In this paper we study the problem of detecting non-sequential access to global shared variables. We employ data flow frameworks in order to solve sub-problems of this general problem. In detail, we set up a data flow framework to find all tasks which potentially run in parallel and we set up a second data flow framework to handle the interprocedural problems of determining variables being "global" to a certain entity. In joining the solutions of these data flow problems, we are able to detect access anomalies in a conservative manner, i.e., if there actually is a non-sequential access to a shared non-protected variable, our approach will detect it. On the other hand, we may also detect false positives.

* Bernd Burgstaller has been supported by the ARC Discovery Project Grant "Compilation Techniques for Embedded Systems" under Contract DP 0560190, and the University of Sydney R&D Grants Scheme "Speculative Partial Redundancy Elimination" under Contract L2849 U3229.

L.M. Pinho and M. González Harbour (Eds.): Ada-Europe 2006, LNCS 4006, pp. 40–55, 2006.

The remainder of the paper is organized as follows. In Section 2 our data flow frameworks to find tasks running in parallel and to determine the set of variables "global" to a program entity are presented. Examples are used to illustrate our framework. In Section 3 we survey related work, before we conclude the paper and describe future work in Section 4.

2 Data Flow Framework

In the following we first define a data flow framework to determine which task objects run in parallel to other task objects. This information is used later on to determine if tasks running in parallel access the same global variables.

2.1 Setting Up Data Flow Equations for Relation ‖

We define a relation ‖ on task objects such that for task objects t_1 and t_2: $t_1 \parallel t_2$ if task objects t_1 and t_2 run in parallel. Note that ‖ commutes, i.e., $t_1 \parallel t_2 \iff t_2 \parallel t_1$.

A control flow graph (CFG) $G = (N, E, r)$ consists of a set N of nodes, and a set $E \subseteq N \times N$ of edges. Node r is the designated root node of the CFG.

Given a $\mathrm{CFG}(t) = (N, E, r)$ of a task body t, the basis for the data flow framework are standard equations [21] of the form

$$\mathrm{S_{out}}(n) = \mathrm{Gen}(n) \cup (\mathrm{S_{in}}(n) \setminus \mathrm{Kill}(n))$$
$$\mathrm{S_{in}}(n) = \bigcup_{n' \in \mathrm{Pred}(n)} \mathrm{S_{out}}(n'),$$

where n denotes a node of a CFG, $\mathrm{Gen}(n)$ is the set of task objects generated (declared or allocated via a new-statement) in node n, and $\mathrm{Kill}(n)$ denotes the set of task objects terminating in node n. All Gen and Kill sets can be empty. Note that Gen sets also include task objects generated indirectly via subprogram or entry calls and via calls of protected operations. Note also that we only take into account the static structure of the underlying multi-tasking program. Thus a task object t is considered to be generated in unit u if there is a statement contained in u that allocates t or u starts with the begin-statement that immediately follows the declarative part containing the declaration of t.

The execution of a handled sequence of statements of a package body is considered to be part of the activation of its *master* task [14]. Thus the CFG of such code is prepended to the CFG of its master task. If there are several such code pieces the corresponding CFGs can be prepended in arbitrary order because if task are generated in these code pieces, their order of activation does not affect the ‖-relation.

Since these are sets a compiler has to determine in order to guarantee that a multi-tasking program is executed correctly, we assume that both Gen and Kill sets can be found automatically (and are available for our analysis).

A detailed description of sets $\mathrm{S_{out}}(n)$ is as follows: If a task object t is generated, it is part of the Gen set, i.e., $t \in \mathrm{Gen}(n)$. If an array of task objects of type

tt is declared, several task objects of type *tt* may run in parallel. We model this by writing $t^* \in \mathrm{Gen}(n)$. In addition, we extend the usual set operations such that $\{t^*\} \cup \{t\} = \{t^*\}$. A similar notation is used for the Kill sets.

We propose to use elimination methods to solve this data flow framework (see e.g. [21] for a survey of elimination methods). As a preliminary step we need a *normal form* for our equations. We note that the equation for $\mathrm{S_{in}}(n)$ can be eliminated by inserting it into $\mathrm{S_{out}}(n)$. From here on we write $\mathrm{S}(n)$ instead of $\mathrm{S_{out}}(n)$ to keep notation short.

Let A, B, C be sets. For set operations "$\cup$" and "$\setminus$" we have

$$(A \cup B) \setminus C = (A \setminus C) \cup (B \setminus C) \tag{1}$$

and

$$(A \setminus B) \setminus C = A \setminus (B \cup C). \tag{2}$$

Let $I \subseteq N$ be a subset of the set of nodes. We define the following normal form for our equations:

$$S(n) = \bigcup_{i \in I} \left(\left(S(i) \setminus \mathrm{Kill}'(i) \right) \cup \mathrm{Gen}'(i) \right).$$

Elimination methods require two rules:

Insertion Rule. One equation has to be substituted into another one. This is straight-forward to do and by repeated application of Eq. (1) and (2) the resulting equation can be brought to normal form again.

Loop Breaking Rule. Given an equation

$$S(n) = \bigcup_{i \in I} \left(\left(S(i) \setminus \mathrm{Kill}'(i) \right) \cup \mathrm{Gen}'(i) \right),$$

where $n \in I$, the task of loop breaking is to find an equivalent equation [20]

$$S'(n) = \bigcup_{i \in I'} \left(\left(S(i) \setminus \mathrm{Kill}''(i) \right) \cup \mathrm{Gen}''(i) \right) \tag{3}$$

such that $n \notin I'$.

We define our loop breaking rule as follows. Let

$$S(n) = \bigcup_{i \in I \setminus \{n\}} \left(\left(S(i) \setminus \mathrm{Kill}'(i) \right) \cup \mathrm{Gen}'(i) \right) \cup \left(\left(S(n) \setminus \mathrm{Kill}'(n) \right) \cup \mathrm{Gen}'(n) \right).$$

Then

$$S'(n) = \left(\left(\bigcup_{i \in I \setminus \{n\}} \left(\left(S(i) \setminus \mathrm{Kill}'(i) \right) \cup \mathrm{Gen}'(i) \right) \right) \setminus \mathrm{Kill}'(n) \right) \cup \mathrm{Gen}'''(n) \tag{4}$$

where $\mathrm{Gen}'''(n) = \{t^* \mid t \in \mathrm{Gen}'(n) \text{ or } t^* \in \mathrm{Gen}'(n)\}$. The definition of Gen''' ensures that t^* is present in the set if task objects t are generated

in the loop body. This implies that task objects of this type are running in parallel if the loop body is executed at least two times. If the loop is iterated less than two times, this is not the case. Since however we do not know the number of loop iterations statically, we assume that the loop is iterated more than once.
Bringing Eq. (4) into normal form yields Eq. (3).

We would like to note that it is not possible to solve our data flow framework via iterative methods [15] because iterative methods require a too simple loop breaking rule.

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

```
CONSTRUCT‖ ()
1   for each task CFG do
2     for each node n do
3       for each t* ∈ S(n) do
4         DEFINE t ‖ t
5       endfor
6       for each pair t1, t2 ∈ S(n) do
7         DEFINE t1 ‖ t2
8       endfor
9     endfor
10  endfor
```

Since our data flow framework is defined on CFGs, and since a multi-tasking program consists of several CFGs (one for each task body), the data flow framework has to be applied to all of them. This, however, has to be done in a certain order because CFG(t) can only be processed if all task bodies corresponding to task objects generated in t have been processed before. This order on task objects can be modeled by a directed acyclic graph (DAG). Thus we apply our data flow framework in reverse topological order (cf. e.g. [19]) to this DAG. The last CFG to be analyzed is that of the environment task (main procedure). Hence, we do not handle mutual task declarations such as those depicted in Fig. 1, even if the declaration is performed only conditionally.

```
1   procedure Mutual is

2     task type B is end B;

3     task type C is end C;

4     task body B is
5       T_Var : C;
6     begin null; end B;

7     task body C is
8       T_Var : B;
9     begin null; end C;

10    The_Task : B;

11  begin null; end Mutual;
```

Fig. 1. Example: Unconditional Mutual Task Declarations

2.2 Determining Sets of Used and Modified Variables

In this section we develop a method to determine the global variables read and written by a task. For this we need the notion of a variable being *global* to a given entity, according to the scoping rules of Ada95. For a declared entity the scope of the declaration denotes those places in the code where it is legal to refer to it. The Ada95 programming language is based on static scoping (cf. [22, p. 176]), which means that visibility of entities at a given program point follows solely from the lexical structure of the program, and not from dynamic aspects (such as the point of invocation of a procedure). Section 8.2 of the language reference manual [14] defines the scope of a declaration; a crucial aspect of these scoping rules is that the scope of a declaration that occurs immediately within the visible part of an outer declaration extends to the end of the scope of the outer declaration.

```
1  with G; use G;

2  procedure Main is
3      Local : Integer := 0;
4  begin
5      P (Global);
6      declare
7          task B is end B;
8          task body B is
9             Another : Integer := 0;
10         begin
11            Global := Global + 1;
12         end B;
13     begin null; end;
14 end Main;
```

```
1  package G is
2      Global : Integer;
3      procedure P (X : in out Integer);
4  end G;

1  package body G is
2      U : Integer;
3      procedure P (X : in out Integer) is
4      begin
5        X := X + 1;
6        declare
7          task C is end C;
8          task body C is
9          begin
10           X := X + 1; U := U + 1;
11         end C;
12       begin null; end;
13     end P;
14 end G;
```

Fig. 2. Example Demonstrating Global and Owned Variables

With Ada95, the following constructs act as scopes: blocks, class subtypes and types, entries, functions, loops, packages, subprograms, protected objects, record types and subtypes, private types, task types and subtypes.

Definition 1. *A declaration is local to a declarative region if the declaration occurs immediately within the declarative region. An entity is local to a declarative region if the entity is declared by a declaration that is local to the declarative region [14, Section 8.1(14)].*

Definition 2. *Given a subprogram, task body, a protected entry, procedure or function, or a dispatching operation u (in the following termed* unit *u), we say that u* owns *an entity e, if e is local to the declarative region of u. In addition, task entries constitute units; they* own *the union of the entities owned by their corresponding accept statements. The ownership relation is reflexive and transitive. Moreover, we extend it to the dynamic case in the sense that u owns*

all entities owned by entities called by u. 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.*

It should be noted that our definition of "globalness" is related to up-level addressing, under incorporation of call-chains.

As an example, consider Fig. 2. Variables `Local` and `Another` are owned by procedure `Main`, variable `Another` is also owned by task `B`, and variables `Global` and `U` are global to `Main`, `B`, `C`, and `P`. In this example the formal parameter `X` of procedure `P` is an alias for variable `Global`; we will treat aliasing in the latter part of this section.

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

Given now two task objects t_1 and t_2 with their corresponding task bodies B_1 and B_2. If $t_1 \parallel t_2$, and the intersection of the corresponding sets $\sigma_{rw}(B_1) \cap \sigma_{rw}(B_2)$ is non-empty, then we are facing a potential conflict. If an entity e from the intersection is global to at least one of the participating task bodies B_1 and B_2, and if e is modified in at least one of the participating task bodies (as opposed to just being used), then the conflict is "real". (We will formalize this condition in Section 2.3.)

We are now faced with the problem of determining for each unit u the corresponding quadruple $\langle \mathcal{O}_r, \mathcal{O}_w, \mathcal{G}_r, \mathcal{G}_w \rangle$. This can be related to interprocedural data flow analysis which is concerned with the determination of a conservative approximation of how a program manipulates data at the level of its call graph. In our case we are interested in the owned and global variables read and written by a given unit. Our problem is *flow-insensitive* as we currently do not incorporate control flow information encountered in a unit; as a consequence, a single read or update operation on a given variable v in a unit u is already sufficient to place v in the respective set of u's quadruple.

It is shown in [8] how alias-free flow-insensitive side-effect analysis can be carried out for procedure call graphs and call-by-reference parameter passing. In [9] it is shown how interprocedural flow-insensitive may-alias information can be factored into this result to account for aliases due to call-by-reference parameter passing, for procedures of arbitrary lexical nesting level. This approach assumes however the absence of pointer aliases. In the following we investigate to what extent [8, 9] apply to Ada95 and how those approaches can be adapted to determine the sought quadruples.

Parameter passing: Ada95 employs two types of parameter passing, namely by copy (aka copy-in/copy-out), and by reference. When a parameter is passed by copy, any information transfer between formal and actual parameter occurs

only before and after execution of the subprogram (cf. [12], [14, 6.2]). From the point of view of our analysis method both parameter passing mechanisms are equivalent, because we screen the source code only at the granularity of whole tasks (and subprograms).

Pointer aliases: [8, 9] does not include pointer aliases. Moreover, the may-alias problem for $k > 1$ level pointers is undecidable (cf. e.g., [5]). Hence we chose a conservative analysis strategy with respect to pointer aliasing which assumes that every entity possibly targeted by a pointer is modified during a procedure call. Due to the induced complexity we had to exclude access to subprogram types altogether from our analysis.

Calculation of $\langle \mathcal{O}_r, \mathcal{O}_w, \mathcal{G}_r, \mathcal{G}_w \rangle$: The only statement aggregation in [8] are *procedures*. In the following we write $\mathrm{GMOD}(p)$ to denote the set of *all* variables that may be modified by an invocation of procedure p. Furthermore, we write $\mathrm{IMOD}(p)$ to denote those variables that may be modified by executing procedure p without executing any calls within it.

In order to compute $\mathrm{GMOD}(p)$, [8] sets up a data flow problem that is based on the procedure call graph and consists of equations of the form

$$\mathrm{GMOD}(p) = \mathrm{IMOD}(p) \cup \Big[\bigcup_{e=(p,q)} b_e\big(\mathrm{GMOD}(q) \cap \mathrm{Nonlocals}(q)\big) \Big]. \tag{5}$$

Therein function b_e maps names from procedure q into names from procedure p according to the name and parameter binding at the call site $e = (p, q)$. Specifically, b_e maps the formal parameters of q to the actual parameters at the call site. The intersection of $\mathrm{GMOD}(q)$ with the set of nonlocal variables $\mathrm{Nonlocals}(q)$ ensures that variables local to q are factored out beforehand.

To compute our sought quadruples for Ada95, we can set up a system of equations similar to Eq. (5). Doing so we split the set GMOD into the sets of owned and global variables, and we move from *procedures* to *units* in terms of statement aggregation. (Hence the procedure call graph becomes a unit call graph.) In this way, $\mathrm{IMOD}(u)$ denotes those variables that may be modified by executing unit u without executing any calls to subprograms or entries within it, and without executing any task objects owned by it. We do not count a modification that is due to an initialization expression of a declaration in the declarative_part (cf. [14, 3.11]) of unit u; this is a measure to reduce false positives and will be explained in Section 2.5.

$$\mathcal{G}'_w(u) = \bigcup_{e=(u,u')} b_e\big(\mathcal{G}_w(u')\big) \tag{6}$$

$$\mathcal{G}_w(u) = \big[\mathrm{IMOD}(u) \cap \mathcal{G}(u)\big] \cup \big[\mathcal{G}'_w(u) \setminus \mathcal{O}(u)\big] \tag{7}$$

$$\mathcal{O}_w(u) = \big[\mathrm{IMOD}(u) \cap \mathcal{O}(u)\big] \cup \Big[\bigcup_{e=(u,u')} \mathcal{O}_w(u') \Big] \cup \big(\mathcal{G}'_w(u) \cap \mathcal{O}(u)\big) \tag{8}$$

Eq. (6) denotes the set of variables which are modified by called units of u and which are global to those called units. In Eq. (7) we determine the set $\mathcal{G}_w$

of unit u, which consists of the locally modified global variables of u and those variables of Eq. (6), which are global to u. Finally, the set $\mathcal{O}_w$ of u consists of the locally modified owned variables of u as well as the modified variables owned by called units and those modified global variables of called units which are owned by u. In replacing IMOD by IUSE as the set of used variables, a system of equations similar to Eq. (6)–(8) can be defined to determine the sets $\mathcal{G}_r$ and $\mathcal{O}_r$.

The sets IMOD(u) and IUSE(u) themselves can be computed by a single linear scan of the statements of u. Therein we do not consider variables which are marked by pragmas Atomic or Volatile, or protected variables, as none of them can give raise to access anomalies. In addition we treat accesses to array components as accesses to the whole array. The same applies to records and their components.

Dispatching operations of tagged types require additional thinking — if we cannot determine the target of a *dispatching call* (cf. [14, 3.9.2]) at compile-time, we have to assume calls to all dispatching operations that might be the target of the dispatching call at run-time.

A further source of complication are generic packages, for which we defer analysis to the point of instantiation.

```
FACTOR_IN (ALIAS, U)
1   for each u ∈ U do
2      FACTOR_SET(u, ALIAS, O_r)
3      FACTOR_SET(u, ALIAS, O_w)
4      FACTOR_SET(u, ALIAS, G_r)
5      FACTOR_SET(u, ALIAS, G_w)
6   endfor
```

```
FACTOR_SET (u, ALIAS, in out S_in)
1   S_out := S_in
2   -- add formal parameter aliases:
3   for each v ∈ Ext_Formals(u) do
4      if v ∈ S_in then
5         S_out := S_out ∪ ALIAS(v, u)
6      endif
7   endfor
8   -- add global variable aliases:
9   for each v ∈ Nonlocals(u) ∩ S_in do
10     S_out := S_out ∪ ALIAS(v, u)
11  endfor
12  S_in := S_out
```

Fig. 3. Algorithm to Factor In Aliasing Information

The data flow problem defined above computes alias-free data flow information. Regarding the example given in Fig. 2, this means that e.g., with task body `C`, we are not aware that the formal parameter `X` of procedure `P` is an alias for variable `Global`[1]. To factor in aliasing information, we employ the interprocedural may-alias analysis method from [9]. Let ALIAS(v, u) denote the set of aliases for variable v within unit u. Due to [9] we can compute ALIAS(v, u), for each formal parameter v and for each global variable v for a unit u. We depict in Fig. 3 how this aliasing information can be factored into our alias-free quadruple-based data flow information; this algorithm is an adaption of an algorithm from [9] to our data flow problem at hand.

[1] An alias from the perspective of our analysis method, which is by necessity insensitive to the copy-in/copy-out parameter passing mechanism of Ada95.

We assume that the driver algorithm FACTOR_IN receives as arguments the sets of aliases (ALIAS) and the units (U) of the program under consideration. For each unit u and each set of its associated quadruple $\langle \mathcal{O}_r, \mathcal{O}_w, \mathcal{G}_r, \mathcal{G}_w \rangle$, FACTOR_IN calls the factoring algorithm FACTOR_SET in order to factor in aliasing information. This algorithm proceeds in two steps. The first loop addresses the set *Ext_Formals* of *extended* formal parameters of u, which consists of all formal parameters visible within u, including those of units that u is nested in, that are not rendered invisible by intervening declarations[2]. In the second loop we add the aliases of variables that are non-local to u. Note that FACTOR_SET only adds aliases to variables that are contained in its input-set $\mathrm{S_{in}}$.

In the following section we define operations on our quadruple-based data flow information which allows us to record information on program variables being read or updated non-sequentially.

2.3 Potential Non-sequential Variable Access

We have shown in Section 2.1 how we can compute relation $\parallel$ in order to determine task objects that may execute in parallel. Moreover, in Section 2.2 we have devised an algorithm to compute the sets of global and owned variables used/modified by a task body.

Let $B(t)$ denote the task body of a task object t; with this notation we regard the environment task also as a task object, with its task body being the main procedure of the program. A variable v is used by a task object t, if v is in the set[3] of read variables of the task body of t, that is, $\mathrm{use}(v,t) \Leftrightarrow v \in \sigma_r(B(t))$. Likewise for modifications of v by t, written as $\mathrm{mod}(v,t) \Leftrightarrow v \in \sigma_w(B(t))$. We have now everything in place to formulate the condition for a potential non-sequential variable access between two task objects t_1 and t_2 which may execute in parallel, that is, $t_1 \parallel t_2$.

Definition 3. *Predicate* $\sigma(t_1, t_2)$ *is* true *if some variable* v *is non-sequentially accessed by task objects* t_1 *and* t_2, false *otherwise. It is formally defined as*

$$\sigma(t_1,t_2) = \bigwedge_{v \in S} \Big[\Big[\big(\mathrm{use}(v,t_1) \wedge \mathrm{mod}(v,t_2)\big) \tag{9}$$

$$\vee \big(\mathrm{mod}(v,t_1) \wedge \mathrm{use}(v,t_2)\big) \tag{10}$$

$$\vee \big(\mathrm{mod}(v,t_1) \wedge \mathrm{mod}(v,t_2)\big) \Big] \tag{11}$$

$$\wedge \big(v \in \sigma_G(B(t_1)) \cup \sigma_G(B(t_2))\big) \Big], \tag{12}$$

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* (12) *ensures that variable* v *is global to at least one of the involved task bodies.*

[2] It is shown in [9] how *Ext_Formals* can be computed from the so-called binding graph of procedure parameters.

[3] Cf. Section 2.2 for the definition of these sets.

Note that $t_1 = t_2$ is not excluded by this definition. In order to see that this is useful consider two tasks of the same task type tt being allocated via new statements (e.g. in a loop-body). Thus we have $t_1 = t_2$, say, and $t_1 \parallel t_2$. Now, if both t_1 and t_2 modify variable v which is locally declared in tt, $\sigma(t_1, t_2)$ evaluates to false only because Eq. (12) becomes false in this case.

2.4 Complexity Issues

The data flow problem described in Section 2.1 can be solved via elimination methods in $O(|E| \cdot \log |N|)$ time [25], where $|N|$ denotes the number of nodes in a CFG and $|E|$ the number of edges in a CFG.

As shown in [8, 9], the data flow problem stated in Section 2.2 can be solved in $O(|E| \cdot |N| + |N|^2)$, with $|N|$ and $|E|$ being the number of call graph nodes and edges.

Summing up, our method performs very efficiently in analyzing Ada multi-tasking programs for detecting access anomalies.

2.5 A Simple Example

For purposes of demonstration we have chosen a simple concurrent Ada program without aliasing effects. It is the well know Producer/Consumer pattern, with its source code depicted in Figure 4. In procedure `Erroneous` (which is also the main subprogram of this example), variable `a` and two tasks, `Producer p` and `Consumer c`, are declared. Both of them are using variable `a` (the producer is even modifying it) ten times in an unsynchronized way.

```
procedure Erroneous is
   a : Integer := 0;                                  -- Node 1
   task type Producer(Count : Natural) is             -- Node 1
   end Producer;                                      -- Node 1
   task type Consumer(Count : Natural) is             -- Node 1
   end Consumer;                                      -- Node 1
   task body Producer (Count : Natural) is
   begin
      for i in 1..Count loop                          -- Node 2
         a := i;                                      -- Node 3
         -- do something else in the meantime          - Node 3
      end loop;
   end Producer;
   task body Consumer (Count : Natural) is
   begin
      for j in 1..Count loop                          -- Node 4
         -- read global variable a                    -- Node 5
      end loop;
   end Consumer;
   p : Producer(10);                                  -- Node 1
   c : Consumer(10);                                  -- Node 1
begin
   null;                                              -- Node 1
end Erroneous;
```

Fig. 4. Example: Source Code

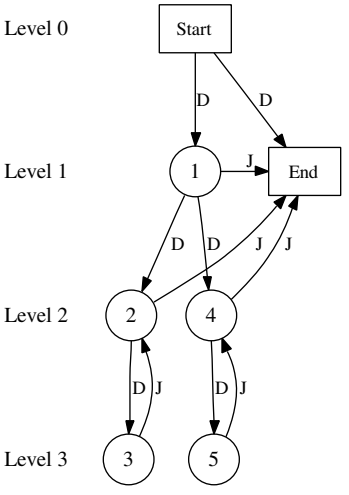

Fig. 5. Example: DJ-Graph

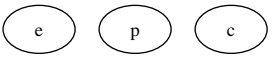

Fig. 6. Example: Unit Call Graph

The data flow equations for the example shown in Figure 5 are set up as follows (for simplicity we abbreviate `Erroneous` by "e"):

$$
\begin{aligned}
S(\text{Start}) &= \{e\},\\
S(1) &= (S(\text{Start})\backslash\text{Kill}(1)) \cup \text{Gen}(1) = (\{e\}\backslash\emptyset) \cup \{p,c\} = \{e,p,c\},\\
S(2) &= ((S(1) \cup S(3))\backslash\text{Kill}(2)) \cup \text{Gen}(2) = S(1) \cup S(2),\\
S(3) &= (S(2)\backslash\text{Kill}(3)) \cup \text{Gen}(3) = S(2),\\
S(4) &= ((S(1) \cup S(5))\backslash\text{Kill}(4)) \cup \text{Gen}(4) = S(1) \cup S(5),\\
S(5) &= (S(4)\backslash\text{Kill}(5)) \cup \text{Gen}(5) = S(4),\\
S(\text{End}) &= ((S(\text{Start}) \cup S(1) \cup S(2) \cup S(4))\backslash\text{Kill}(\text{End})) \cup \text{Gen}(\text{End})\\
&= S(\text{Start}) \cup S(1) \cup S(2) \cup S(4).
\end{aligned}
$$

We employ the eager elimination method due to [25] to solve the data flow equations of our example. This method is based on DJ graphs, the union of a CFG and its dominator tree (cf. [25]). It requires to distinguish between d- and j-edges (cf. Fig. 5). For details the reader is referred to [25]. First the bottom-up join edge elimination phase (and simultaneous insertion in the data flow equations) of the eager elimination method is started at level 3: $5 \rightarrow 4$: $S(4) = S(1) \cup S(4)$; $3 \rightarrow 2$: $S(2) = S(1) \cup S(2)$.
At level 2 loop breaking is necessary: $\not\emptyset$ 4: $S(4) = S(1)$, $\not\emptyset$ 2: $S(2) = S(1)$. During the second phase of the eager elimination method the solution is propagated along d-edges in a top down manner: $S(2) = S(3) = S(4) = S(5) = \{e,p,c\}$; $S(\text{End}) = \{e,p,c\}\backslash\{e,p,c\} = \emptyset$.

According to the algorithm for constructing the $\|$-relation from Section 2.1, we get $e \parallel p$, $e \parallel c$, and $p \parallel c$.

In the following e, p, and c denote the nodes of the unit call graph of our example, which is depicted in Fig. 6. (Note that since our simple example does not contain any calls, the unit call graph is in fact trivial). According to the data flow framework given in Section 2.2 we obtain the sets

$$
\begin{aligned}
\mathcal{O}(p) &= \{i\},\\
\mathcal{O}(c) &= \{j\},\\
\mathcal{O}(e) &= \{a,i,j,p,c\}.\\
\mathcal{G}'_w(p) &= \emptyset,\\
\mathcal{G}_w(p) &= \big[\mathrm{IMOD}(p) \cap \mathcal{G}(p)\big] \cup \big[\mathcal{G}'_w(p) \setminus \mathcal{O}(p)\big] = \big[\{a,i\} \cap \{a\}\big] \cup \emptyset = \{a\},\\
\mathcal{O}_w(p) &= \big[\mathrm{IMOD}(p) \cap \mathcal{O}(p)\big] \cup \emptyset \cup (\mathcal{G}'_w(p) \cap \mathcal{O}(p))\\
&= \big[\{a,i\} \cap \{i\}\big] \cup \emptyset \cup (\emptyset \cap \{i\}) = \{i\},\\
\mathcal{G}'_w(c) &= \emptyset,\\
\mathcal{G}_w(c) &= \big[\mathrm{IMOD}(c) \cap \mathcal{G}(c)\big] \cup \big[\mathcal{G}'_w(c) \setminus \mathcal{O}(c)\big] = \big[\{j\} \cap \{a\}\big] \cup \emptyset = \emptyset,\\
\mathcal{O}_w(c) &= \big[\mathrm{IMOD}(c) \cap \mathcal{O}(c)\big] \cup \emptyset \cup (\mathcal{G}'_w(c) \cap \mathcal{O}(c)) = \big[\{j\} \cap \{j\}\big] \cup \emptyset = \{j\},\\
\mathcal{G}'_w(e) &= \emptyset,\\
\mathcal{G}_w(e) &= \big[\mathrm{IMOD}(e) \cap \mathcal{G}(e)\big] \cup \big[\mathcal{G}'_w(e) \setminus \mathcal{O}(e)\big] = \big[\emptyset \cap \emptyset\big] \cup \big[\emptyset \setminus \{a,i,j,p,c\}\big] = \emptyset,\\
\mathcal{O}_w(e) &= \big[\mathrm{IMOD}(e) \cap \mathcal{O}(e)\big] \cup (\mathcal{G}'_w(e) \cap \mathcal{O}(e))\\
&= \big[\emptyset \cap \{a,i,j,p,c\}\big] \cup (\emptyset \cap \{a\}) = \emptyset.\\
\mathcal{G}'_r(p) &= \emptyset,\\
\mathcal{G}_r(p) &= [\mathrm{IUSE}(p) \cap \mathcal{G}(p)] \cup [\mathcal{G}'_r(p) \setminus \mathcal{O}(p)] = [\{i\} \cap \{a\}] \cup \emptyset = \emptyset,\\
\mathcal{O}_r(p) &= [\mathrm{IUSE}(p) \cap \mathcal{O}(p)] \cup \emptyset \cup (\mathcal{G}'_r(p) \cap \mathcal{O}(p)) = [\{i\} \cap \{i\}] \cup (\emptyset \cap \{i\}) = \{i\},\\
\mathcal{G}'_r(c) &= \emptyset,\\
\mathcal{G}_r(c) &= [\mathrm{IUSE}(c) \cap \mathcal{G}(c)] \cup [\mathcal{G}'_r(c) \setminus \mathcal{O}(c)] = [\{a,j\} \cap \{a\}] \cup \emptyset = \{a\},\\
\mathcal{O}_r(c) &= [\mathrm{IUSE}(c) \cap \mathcal{O}(c)] \cup \emptyset \cup (\mathcal{G}'_r(c) \cap \mathcal{O}(c))\\
&= [\{a,j\} \cap \{j\}] \cup (\emptyset \cap \{j\}) = \{j\},\\
\mathcal{G}'_r(e) &= \emptyset,\\
\mathcal{G}_r(e) &= [\mathrm{IUSE}(e) \cap \mathcal{G}(e)] \cup [\mathcal{G}'_r(e) \setminus \mathcal{O}(e)] = \emptyset \cup [\emptyset \setminus \{a,i,j,p,c\}] = \emptyset, \text{ and}\\
\mathcal{O}_r(e) &= [\mathrm{IUSE}(e) \cap \mathcal{O}(e)] \cup (\mathcal{G}'_r(e) \cap \mathcal{O}(e)) = \emptyset \cup (\emptyset \cap \{a,i,j,p,c\}) = \emptyset.
\end{aligned}
$$

As already mentioned in Section 2.2, we do not count a modification that is due to an initialization expression of a declaration in the declarative part of unit u. This is justified by the fact that declarations in declarative part D are (1) not visible/accessible outside the scope of this task, and (2) the elaboration order ensures that tasks declared in D are activated after the declaration and initialization of the variables in D. This effectively serializes the modifications due to initialization with possible accesses from within child tasks. Thus in our example variable a is not a member of $\mathrm{IMOD}(e)$.

Furthermore we get $\sigma_{rw}(e) = \emptyset$, $\sigma_{rw}(p) = \{a,i\}$, and $\sigma_{rw}(c) = \{a,j\}$. We have now $\sigma_{rw}(e) \cap \sigma_{rw}(p) = \emptyset$, and $\sigma(e,p) = \mathit{false}$. Because of $\sigma_{rw}(e) \cap \sigma_{rw}(c) = \emptyset$

and $\sigma(e, c) = \mathit{false}$, the same applies to tasks e and c. From $\sigma_{rw}(p) \cap \sigma_{rw}(c) = \{a\}$ and $\sigma(p, c) = \mathit{true}$ we conclude that there is an access anomaly concerning tasks c and p with respect to variable `a`.

3 Related Work

In [10, 18, 17] a detailed survey of possible erroneous executions in Ada (especially unsynchronized accesses to unprotected variables and how unpredictable the results are) is presented. Although there are protected types in Ada 95, unprotected variables can be and are used. *"... we do not wish to jump to the simple conclusion that unprotected non-local variables should not be used. ... although the need for them has now been greatly reduced ... perform a mechanical verification of the fact that they are used correctly"* [17].

One way to cope with unpredictability is to allow just a strict (safe) subset of the Ada programming language [7, 4]. The Ravenscar Profile [6] removes non-deterministic tasking features from Ada and thus provides a statically analyzable subset of tasking facilities of Ada 95. This enables the development of high-integrity systems even in conjunction with tasks. *"The avoidance of unprotected shared variables is generally a requirement of high integrity systems, although detection of this erroneous case is not mandated by the Ravenscar Profile definition"* [7]. Thus, even in combination with the Ravenscar Profile, an additional check is needed to make sure that unprotected data is never shared between tasks. The Ravenscar Profile is an opportunity to allow concurrency within SPARK [4, 1].

A variety of approaches dealing with the detection of tasking anomalies in multi-tasking (Ada) programs have been proposed. These approaches include *static analysis*, *post-mortem trace analysis*, *on-the-fly monitoring*, and combinations. In [13] an overview of available techniques is presented. The goal of static analysis is to detect access anomalies prior to execution. On-the-fly monitoring is a dynamic approach and usually combined with a debugging tool. Post-mortem methods include all techniques used to discover errors in an execution following its termination.

Static Concurrency Analysis, presented in [26], is a method for determining concurrency errors in parallel programs. The class of detectable errors includes infinite waits, deadlocks, and concurrent updates of shared variables. Potentially concurrent sections of code are identified. Shared variable operations in these sections are potential anomalies. The algorithm is however exponential in the number of tasks in the program.

Detecting access anomalies by monitoring program execution is proposed in [23]. A general on-the-fly algorithm is presented, which can be applied to programs containing both nested fork-join and synchronization operations. In [11] the dynamic approach is further explored, nested parallel loops are considered, and experimental results are given. The retrospective in [24] gives a good survey of on-the-fly techniques. In general these techniques are fundamentally different to our static analysis approach. To reduce the amount of run-time checking,

static program analysis can be used in combination with an on-the-fly approach (cf. e.g., [13]).

AdaWise [3] is a set of program analysis tools that performs automatic checks to verify the absence of common run-time errors affecting correctness or portability of Ada code. AdaWise checks at compile-time for potential errors such as incorrect order dependence or erroneous execution due to improper aliasing. Like our approach, it operates in a conservative way. That is, the absence of a warning guarantees the absence of a problem. If AdaWise produces a warning, there is a potential error that should be investigated by the developer.

A good survey of available tools detecting races in Java (e.g. rccjava, Java Pathfinder, ESC/Java, Bandera) or C (e.g. Warlock and RacerX) can be found in [27].

4 Conclusion and Future Work

In this paper we have presented data flow analysis frameworks for detecting non-sequential access of shared non-protected variables, so-called access anomalies. Our framework can handle most programs of practical importance. It is computationally efficient and easy to implement by modifying the source code of an existing compiler like GNAT. Toolkits for constructing data flow analyzers [16] can also be employed. Our method is conservative and may therefore raise false positives. It should be easily adaptable for the Ravenscar profile [6, 7].

Our approach is also well-suited for other programming languages like Java [2], although a Java program is not even termed erroneous if it accesses global shared variables in a non-sequential way.

In the future we plan to develop a symbolic analysis framework that is aimed at the detection of non-sequential global shared variable access. Symbolic analysis is capable of incorporating flow-sensitive side-effects of a program, which will make it less susceptible to the detection of false positives. A refinement of relation $\|$ to model parallelism in a more fine-grained (i.e., intra-task) manner is an orthogonal measure to reduce the number of false positives. At the moment our analysis considers parallelism only on a per-task basis, which is a safe approximation of the actual potential for parallelism between variable accesses. There are however many cases where task objects executing in parallel access a common variable, but the intra-task structure of the program reveals that the actual access operations cannot occur in parallel (e.g., due to involved synchronization primitives).

References

1. SPARK Examiner, The SPARK Ravenscar Profile, http://www.praxis-his.com/sparkada/pdfs/examiner_ravenscar.pdf, 2004.
2. K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language.* Addison-Wesley, Reading, MA, 3rd edition, 2000.

3. C. Barbasch and D. Egnor. Always one more bug: applying AdaWise to improve Ada code. In *Proceedings of the conference on TRI-Ada '94*, pages 228–235, New York, NY, USA, 1994. ACM Press.
4. J. Barnes. *High Integrity Software - The SPARK Approach to Safety and Security.* Addison-Wesley, Harlow, England, 2003.
5. J. Blieberger, B. Burgstaller, and B. Scholz. Interprocedural Symbolic Evaluation of Ada Programs with Aliases. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 136–145, Santander, Spain, June 1999.
6. A. Burns. The Ravenscar Profile. *Ada Lett.*, XIX(4):49–52, 1999.
7. A. Burns, B. Dobbing, and T. Vardanega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. *Ada Lett.*, XXIV(2):1–74, 2004.
8. K. D. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 57–66, New York, NY, USA, 1988. ACM Press.
9. K. D. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–59, 1989.
10. P. Delrio and F. Mazzanti. The risk of destructive run-time errors. *Ada Lett.*, XI(1):102–113, 1991.
11. A. Dinning and E. Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *PPOPP '90: Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 1–10, New York, NY, USA, 1990. ACM Press.
12. W. Gellerich and E. Ploedereder. Parameter-induced aliasing and related problems can be avoided. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, pages 161–172, 1997.
13. R. Hood, K. Kennedy, and J. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 74–81, Washington, DC, USA, 1990. IEEE Computer Society.
14. ISO/IEC 8652. *Ada Reference manual*, 1995.
15. G. Kildall. A unified approach to global program optimization . In *Proc. of the First ACM Symposium on Principles of Programming Languages*, pages 194–206, New York, NY, 1973.
16. J. H. E. F. Lasseter. Toolkits for the automatic construction of data flow analyzers. Technical report, "University of Oregon, Computer & Information Sci. Dept.", 2005.
17. C. Marzullo and F. Mazzanti. Towards the static detection of erroneous executions in Ada 95. Technical report, "Ninth International Software Quality Week '96 (QW'96)", Sheraton Palace Hotel, San Francisco, California USA, 1996.
18. F. Mazzanti. Guide to erroneous executions in Ada 95. Technical report, Centre National de la Recherche Scientifique, Paris, France, France, 1997.
19. K. Mehlhorn. *Graph Algorithms and NP-Completeness*, volume 2 of *Data Structures and Algorithms*. Springer-Verlag, Berlin, 1984.
20. M. C. Paull. *Algorithm Design – A Recursion Transformation Framework*. Wiley Interscience, New York, NY, 1988.
21. B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, 1986.
22. D. A. Schmidt. *Denotational Semantics — A Methodology for Language Development*. Allyn and Bacon, 1986.

23. E. Schonberg. On-the-fly detection of access anomalies. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, pages 285–297, New York, NY, USA, 1989. ACM Press.
24. E. Schonberg. On-the-fly detection of access anomalies. *SIGPLAN Not.*, 39(4): 313–327, 2004.
25. V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. A new framework for elimination-based data flow analysis using DJ graphs. *ACM Trans. Program. Lang. Syst.*, 20(2): 388–435, 1998.
26. R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Commun. ACM*, 26(5):361–376, 1983.
27. F. Zhou. Survey: Race Detection and Atomicity Checking, CS263 Course Project, 2003.