# Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs

Johann Blieberger[1], Bernd Burgstaller[1], and Bernhard Scholz[2]

[1] Department of Computer-Aided Automation
Technical University of Vienna
{blieb,bburg}@auto.tuwien.ac.at
[2] Institute of Computer Languages
Technical University of Vienna
scholz@complang.tuwien.ac.at

**Abstract.** It is well accepted that designing and analyzing concurrent software-components are tedious tasks. Assuring the quality of such software requires formal methods, which can statically detect deadlocks. This paper presents a symbolic data flow analysis framework for detecting deadlocks in Ada programs with tasks. The symbolic data flow framework is based on symbolic evaluation – an advanced technique to statically determine properties of programs.
The framework can guarantee the deadlock-freeness for an arbitrary hardware environment. Our approach differs from existing work in that tasks can be dynamically created and completed in the program. Examples are used to illustrate our approach.

## 1 Introduction

Modern software design includes concurrent programming, such as tasks, to enable the explicit expression of parallelism. Concurrent language constructs increase the expressiveness of a language in the description of concurrency. However, it is widely agreed that designing and programming a concurrent system are tedious tasks and can result in erroneous program behavior. Such anomalies are particular hard to detect as it is usually very difficult to reproduce them and to identify the source of the error. Clearly, it is of paramount importance to detect program anomalies in concurrent systems at compile time as compared to expensively testing programs at runtime. A static analysis tool that targets the detection of concurrent program anomalies supports the design and programming and improves the quality of the software.

In this paper we introduce a new static analysis framework to detect deadlocks and other tasking anomalies without executing the program. Our static analysis is based on symbolic evaluation – an advanced static analysis technique – in which symbolic expressions are used to denote the values of program variables and computations and a path condition describes the impact of the program's control flow onto the values of variables.

Our method goes beyond previous work in the following points:

1. The number of tasks in the program is not limited, it can grow dynamically during the execution of the tasking program. We can handle an arbitrary number of tasks, which can be dynamically created and completed in the program.
2. We analyze each task body only once, although several instances of the task can be created either statically (e.g. arrays of tasks) or dynamically by new statements.
3. Our framework correctly analyzes generic units.

To ease the analysis we assume that no global variable is read/written by different tasks (no shared variables). If such a variable is needed, a protected object has to be employed.

The remainder of the paper is organized as follows. In Section 2 we give the notions of the symbolic data flow analysis for detecting deadlocks. In Section 3 our approach is presented. Examples are used to illustrate our framework. In Section 4 we survey related work. In Section 5 we conclude this paper and describe future work.

## 2 Symbolic Evaluation

*Symbolic evaluation* is an advanced static program analysis in which symbolic expressions are used to denote the values of program variables and computations (cf. e.g. [6]). A path condition describes the impact of the program's control flow onto the values of variables and the condition under which control flow reaches a given program point. In the past symbolic evaluation has been successfully applied to the reaching definitions problem [2], to worst-case execution time analysis [1], to cache hit prediction [4], to alias analysis [3], to optimization problems of High-Performance Fortran [12], and to pointer analysis for detecting memory leaks [17].

The underlying program representation for symbolic evaluation is the *control flow graph (CFG)*, a directed labelled graph. Its nodes are basic blocks containing the program statements, whereas its edges represent transfers of control between basic blocks. Each edge of the CFG is assigned a condition which must evaluate to true for the program's control flow to follow this edge. *Entry* and *Exit* are distinguished nodes used to denote start and terminal node.

In the center of our symbolic analysis is the *program context*, which includes *states* $S_i$ and *state conditions* $C_i$. A program context completely describes the variable bindings at a specific program point together with the associated state conditions and is defined as $\bigcup_{i=1}^{k}[S_i, C_i]$, where $k$ denotes the number of different program states. State $S$ is represented by a set of pairs $\{(v_1, e_1), \ldots, (v_m, e_m)\}$ where $v_i$ is a program variable, and $e_i$ is a symbolic expression describing the value of $v_i$ for $1 \leq i \leq m$. For each variable $v_i$ there exists exactly one pair $(v_i, e_i)$ in $S$. A state condition $C_i$ specifies a condition that is valid for a given state $S_i$ at a certain program point.

For all nodes in the CFG, a set of symbolic equations is used to compute program contexts. The equation system is solved by an elimination algorithm for data flow analysis [16].

## 3 Symbolic Data Flow Equations for Tasking Programs

In previous work [3, 17, 4, 1] we have applied symbolic analysis for sequential programs. In order to analyze Ada programs with tasks we need a new form of analysis to cope with these notions of parallelism. To get a handle on the problem a new program representation, namely the *tasking control flow graph* (TCFG), is introduced which models the semantic behavior of concurrent Ada programs. In the following, semantics and the construction of TCFGs are discussed. Then, the symbolic equations for the program contexts are given and illustrated by two examples.

To symbolically analyze Ada programs with tasks, we are confronted with three major problems to model tasking correctly:

1. How do we handle dynamically allocated tasks?
2. If several entry calls are served by one accept statement, how can we ensure that control flow proceeds at the correct place?
3. According to the semantics of Ada tasks, a task is allowed to complete only if all its descendants have completed or are "waiting" at a select statement with a terminate alternative. How do we ensure this behavior?

Problem 1 is solved by introducing symbolic *task identifiers* (task ids). Each task object is assigned a unique task id at creation time. Such a task id can in its simplest way be realized as a symbolic counter, that is incremented whenever a task is created. (In fact we propose a different counter for each task type, but this does not show up in the examples.) All local variables of a task object and other local objects such as even tasks or protected objects can be referenced uniquely with help of this task id. We denote such objects by $v[id]$, which are modelled as symbolic arrays [3, 4].

Concerning problem 2 we notice that entry calls are different from procedure calls because tasks have a *state*. A procedure call can be modelled by a suitable copy of the procedure with actual parameters supplied accordingly [3]. In contrast, we solve Problem 2 by introducing a symbolic variable $E_n[id]$ for accept statement $n$ which is indexed by the task id of the calling task. The variable is assigned the value $r$ if this is the $r$th entry call to $n$. We ensure that the control flow that enters $n$ is propagated to the correct successor after the rendezvous by assigning condition $E_n[id] = r$ to the control flow from the end of the rendezvous to the correct successor.

In order to solve Problem 3, we introduce variable $T_t$ if one or more instances of task $t$ are created in statement $d$. The purpose of $T_t$ is similar to that of $E_n[id]$ above in ensuring that if several tasks are created at different places in the program, their completion is awaited at the correct place. In addition, for each created task (with task identifier equal to $id$) we introduce a symbolic variable $C_{d,t}[id]$ which is initialized to 1. If the task completes or may complete because of a terminate alternative, this variable is set to 0 and the condition $C_{d,t}[id] = 0$ is checked at an appropriate program point to ensure that each created task ultimately completes.

### 3.1 Building the CFG Tasking Forest

To build the TCFG, for each task body and for the main program a CFG is constructed at first. The result is a *CFG tasking forest*.

1. In a task body each accept statement is a subgraph of the (task) CFG with one designated "header node".
2. There may be several (at least one) "end nodes" of a rendezvous according to the construction of a CFG.
3. A select statement of the callee is modelled like a case statement with the only difference that all conditions from the "header" to the "cases" are *true*. Select statements with a terminate alternative are described in Section 3.2.
4. Select statements of the caller are modelled like if-statements.

### 3.2 Building the Tasking CFG from the CFG Tasking Forest

Then, based on the CFG tasking forest we can construct the TCFG. Note that if ordinary control flow is indicated by a TCFG edge, we call this edge *control flow edge*. If the edge of the TCFG represents an entry call, an entry return, task creation or task completion, we call the edge *tasking edge*. Furthermore, we distinguish between 5 kinds of TCFG nodes according to the following criteria:

1. If no tasking edge points to node $n$, we call $n$ "control flow node".
2. One or more tasking edges point to node $n$ which models an accept statement. Then $n$ is called "header node" of a rendezvous.
3. One or more tasking edges point to the "start node" of a task, which is equal to the entry node of the CFG modelling the corresponding task body.
4. One or more tasking edges point from the end nodes of a rendezvous to the successor nodes of an entry call node ("entry successor node").
5. One or more tasking edges point from the original exit node of the task CFG to the exit node of the parent task. We call the latter exit node "synchronizing node" because all dependent tasks synchronize at this node before the parent task is allowed to complete.

The CFG tasking forest is now glued together to create the TCFG, which models the whole tasking program. This is done by inserting (dashed) tasking edges between certain nodes of the CFGs forming the forest and by removing certain (solid) control flow edges. The TCFG is build according to the following rules:

- For node $n$ being the target of a tasking-related action:
  - Let $id$ denote the task id of the calling task which in its simplest way can be realized by incrementing a symbolic counter each time a task is created statically or dynamically. Introduce variable $E_n[id]$ and initialize it to 0 when the called task is created. Set the value of $E_n[id]$ to $r$ if this is the $r$th tasking edge pointing to $n$.
  - Insert a tasking edge (a dashed arrow) from the caller to the corresponding start node of the rendezvous.
  - Insert a tasking edge from all end nodes of the rendezvous to all entry successors nodes and assign the condition $(E_n[id] = r)$ to these edges.

  - Remove the control flow edges from the entry call node to its (control flow) successors.
- If one or more instances of task $t$ are created in node $d$:
  - Introduce the variables $T_t$ and for each created task $C_{d,t}[id]$ ($T_t$ is initialized to 0, $C_{d,t}[id]$ to 1).
  - Create an intermediate node $i_{dt}$ and insert tasking edges from $d$ to $i_{dt}$ and from $i_{dt}$ to $t$. In addition, node $i_{dt}$ is used to assign $r$ to $T_t$ if this is the $r$th tasking edge pointing to $t$.
  - Create an intermediate node between each exit node of task $t$ and the synchronizing node of its parent. Similarly, create an intermediate node between all headers of a select statement with a terminate alternative and the synchronizing node of its parent.
  - Insert tasking edges to and from the intermediate node. In addition, the intermediate node is used to assign 0 to $C_{d,t}[id]$, the edge leading to the intermediate node is assigned the condition $(T_t = r)$ and the edge leaving the intermediate node is assigned the condition $(C_{d,t}[id] = 0)$.

### 3.3 Setting Up the Data Flow Equations

The symbolic equation system of concurrent Ada programs is given in this subsection. The equations are derived from the TCFG.

Let $C^c_{n' \to n}$ denote the symbolic condition of control flow edge $e^c = n' \to n$ and similarly $C^t_{n' \to n}$ the symbolic condition of tasking flow edge $e^t = n' \to n$. Furthermore, we denote the control flow predecessors of node $n$ by $\mathrm{Pred}^c(n)$ and the tasking flow predecessors by $\mathrm{Pred}^t(n)$. We define the symbolic equations as follows:

1. If node $n$ is a control flow node, we have

$$X_n = \left( \bigvee_{n' \in \mathrm{Pred}^c(n)} (X_{n'} \wedge C^c_{n' \to n}) \right) \mid \{\ldots\}.$$

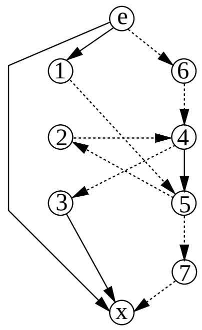
2. If node $n$ is a header node, a start node, or an entry successor node, we define

$$X_n = \left( \bigvee_{n' \in \mathrm{Pred}^c(n)} (X_{n'} \wedge C^c_{n' \to n}) \right) \wedge \left( \mathrm{false} \vee \bigvee_{n' \in \mathrm{Pred}^t(n)} \left(X_{n'} \wedge C^t_{n' \to n} \wedge \mathrm{Guard}(n)\right) \right) \mid \{\ldots\},$$

   where $\mathrm{Guard}(n)$ denotes a guard condition of an accept statement. $\mathrm{Guard}(n)$ is considered to be true if no guard condition is present or in case of start or entry successor nodes.
   Note that if $n$ is a header node and there is no entry call for this entry, then $X_n = \mathrm{false}$.

```
procedure Simple is
   task T1 is
      entry One;
      entry Two;
   end T1;
   task body T1 is
   begin
      accept One;              -- Node 4
      accept Two;              -- Node 5
   end T1;
begin
   T1.Two;                     -- Node 1
   T1.One;                     -- Node 2
end Simple;                    -- Node 3
```

(a) Source Code




(b) TCFG

**Fig. 1.** Simple Deadlock Example

3. If node $n$ is a synchronizing node, we have

$$X_n = \left( \bigvee_{n' \in \mathrm{Pred}^c(n)} (X_{n'} \wedge C^c_{n' \to n}) \right) \wedge \left( \bigwedge_{n' \in \mathrm{Pred}^t(n)} (X_{n'} \wedge C^t_{n' \to n}) \right) \mid \{\ldots\},$$

The $\{\ldots\}$-part is supposed to contain the local changes to program variables in the same way as is described in [1–3].

Note that $X_n$ means "non-blocking" at node $n$. If, after solving the equations and binding the – until now unbound – task identifiers of task $t$ ($id_t$) by prepending $\bigwedge_t \bigwedge_{id_t}$ to all conditions, all $X_n$ evaluate to true, then there is no deadlock in the tasking program. If some of the conditions do not evaluate to true, it has to be checked by hand whether

- there is a deadlock,
- there are some program paths with a false condition because the program is not supposed to terminate (this is typical for embedded systems), or
- there are some accept statements which are never called in the program.

### 3.4 A Simple Example

For sake of demonstration we have chosen a fairly simple Ada program with one task. Figure 1(a) lists the source code. The task consists of two accept statements. In the main program the two entry calls **Two** and **One** are invoked in subsequent order. In Figure 1(b) the TCFG of the Ada source is shown. Nodes 1, 2, 3, 4, and 5 correspond to statements of the Ada program. Nodes 6 and 7 are intermediate nodes introduced to model tasking correctly. Based on the semantic rules given in Subsection 3.1 and 3.2 artificial variables ($C_{\mathrm{e},4}[1]$, $T_4$, $E_4[1]$, and $E_5[1]$) are introduced and the TCFG of Figure 1(b) is amended by the following additional conditions and statements:

- $C_{\mathrm{e},4}[1] := 1$ in node $e$ due to the fact that one task with task id 1 is created.
- $C_{\mathrm{e},4}[1] := 0$ in node 7 since the task is completed in this node.

- $E_5[1] := 1$ in node 1 and $E_4[1] := 1$ in node 2 represent call entries of node 1 and 2 respectively.
- Except task edge $4 \to 3$, $5 \to 2$, $5 \to 7$, and $7 \to \mathrm{x}$ the conditions of the task edges are set to true.
- Task edge $5 \to 2$ and task edge $4 \to 3$ get the condition $E_5[1] = 1$ and $E_4[1] = 1$, respectively.
- Task edge $5 \to 7$ gets assigned the condition $T_4 = 1$.
- Task edge $7 \to \mathrm{x}$ is conditioned by $C_{\mathrm{e},4}[1] = 0$ because task 1 is supposed to complete here.

The symbolic equations of the example are set up by the rules in Subsection 3.3 as follows,

$$
\begin{aligned}
X_{\mathrm{e}} &= \mathrm{true} \mid \{(T_4, 0), (C_{\mathrm{e},4}[1], 1)\},\\
X_1 &= X_{\mathrm{e}} \mid \{(E_5[1], 1)\},\\
X_2 &= (E_5[1] = 1) \wedge X_5 \mid \{(E_4[1], 1)\},\\
X_3 &= (E_4[1] = 1) \wedge X_4,\\
X_4 &= X_2 \wedge X_6,\\
X_5 &= X_1 \wedge X_4,\\
X_6 &= X_{\mathrm{e}} \mid \{(T_4, 1), (E_4[1], 0), (E_5[1], 0)\},\\
X_7 &= (T_4 = 1) \wedge X_5 \mid \{(C_{\mathrm{e},4}[1], 0)\},\\
X_{\mathrm{x}} &= (C_{\mathrm{e},4}[1] = 0) \wedge X_7 \wedge ((\mathrm{false} \wedge X_{\mathrm{e}}) \vee X_3).
\end{aligned}
$$

The equation system is solved step by step. First, we insert $2 \to 4$ and $6 \to 4$ and obtain

$$
X_4 = ((E_5[1] = 1) \wedge X_5 \mid \{(E_4[1], 1)\}) \wedge X_{\mathrm{e}} \mid \{(T_4, 1), (E_4[1], 0), (E_5[1], 0)\},
$$

which evaluates to false. Hence we also get $X_2 = X_3 = X_5 = X_7 = \mathrm{false}$. This proves that node 2 of the example cannot be reached, and a deadlock occurs in the given program.

### 3.5 Modelling Protected Objects

Protected objects are semantically modelled as tasks. The corresponding task body consists of an endless loop containing a select statement with a terminate alternative. The select statement contains one accept statement for each protected operation. Guards are mapped from protected entries to task entries. Protected procedures and functions are mapped to task entries, whereby function return values are (conceptually) replaced with out parameters.

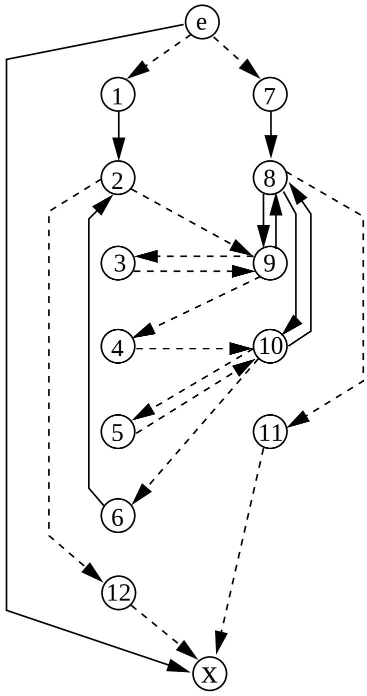
Entry families can easily be integrated in our approach. The same applies to task and entry attributes and to pragma Atomic.

### 3.6 Dining Philosophers Example

The second example is the well-known problem of the *Dining Philosophers*. The source code of an Ada specification and implementation are shown in Figure 2(a).

```
generic
   type Num is mod <>;
procedure Dining;

procedure Dining is

   Current_Id : Num := Num'Last;

   function Next_One return Num is
   begin
      Current_Id := Current_Id + 1;
      return Current_Id;
   end Next_One;

   task type Philosopher(Id: Num := Next_One);

   protected type Fork is
      entry Seize;
      procedure Release;
   private
      Seized : boolean := false;
   end Fork;

   type Phil_Array is array (Num'Range) of
      Philosopher;
   type Fork_Array is array (Num'Range) of Fork;

   Phils: Phil_Array;                          -- Node 1
   Forks: Fork_Array;                          -- Node 1

   task body Philosopher is
      dur: duration;
   begin
      loop                                     -- Node 2
         Forks(Id).Seize;                      -- Node 3
         Forks(Id+1).Seize;                    -- Node 4
         Forks(Id).Release;                    -- Node 5
         Forks(Id+1).Release;                  -- Node 6
      end loop;
   end Philosopher;

   protected body Fork is    -- loop -- Node 7, 8
      entry Seize when not Seized is           -- Node 9
      begin
         Seized := true;                       -- Node 9
      end Seize;
      procedure Release is                     -- Node 10
      begin
         Seized := False;                      -- Node 10
      end Release;
   end Fork;                                   -- end loop;

begin
   null;
end Dining;
```

(a) Source Code




(b) TCFG

**Fig. 2.** Dining Philosophers Example

The corresponding TCFG is depicted in Figure 2(b) where nodes 1 to 6 originally are part of the task *Philosopher* and nodes 7 to 10 are part of the protected type *Fork*. Control flow edges between nodes 2 and 3, nodes 3 and 4, nodes 4 and 5, and nodes 5 and 6 have been removed. Nodes 11 and 12 are intermediate nodes. To keep the example small, the intermediate nodes $i_{\mathrm{e},1}$ and $i_{\mathrm{e},7}$ have been omitted, since there is only one tasking edge to node 1 and to node 7, respectively. In addition, we do not model function `Next_One` which is used to obtain unique task identifiers (in the Ada program and not in its symbolically evaluated version). To ease the readability, arrays are denoted in an Ada-like notation instead of using the symbolic notation (compare [3, 4]). The data flow equations have the following form:

$$\begin{aligned}
X_{\mathrm{e}} &= \mathrm{true} \mid \{(C_{\mathrm{e},1}[0..N-1],1),(C_{\mathrm{e},7}[0..N-1],1),(\mathit{Seized}[0..N-1],\mathrm{false})\}\\
X_1 &= X_{\mathrm{e}}\\
X_2 &= X_1 \vee X_6 \mid \{(E_9[\mathit{id}],1)\}\\
X_3 &= (E_9[\mathit{id}]=1) \wedge X_9 \mid \{(E_9[\mathit{id}],2)\}\\
X_4 &= (E_9[\mathit{id}]=2) \wedge X_9 \mid \{(E_{10}[\mathit{id}],1)\}\\
X_5 &= (E_{10}[\mathit{id}]=1) \wedge X_{10} \mid \{(E_{10}[\mathit{id}],2)\}\\
X_6 &= (E_{10}[\mathit{id}]=2) \wedge X_{10}\\
X_7 &= X_{\mathrm{e}} \mid \{(E_9[0..N-1],0),(E_{10}[0..N-1],0)\}\\
X_8 &= X_7 \vee X_9 \vee X_{10}\\
X_9 &= X_8 \wedge [(X_2 \wedge \neg \mathit{Seized}[\mathit{id}] \mid \{(\mathit{Seized}[\mathit{id}],\mathrm{true})\}) \vee\\
&\qquad\qquad (X_3 \wedge \neg \mathit{Seized}[\mathit{id}-1] \mid \{(\mathit{Seized}[\mathit{id}-1],\mathrm{true})\})]\\
X_{10} &= X_8 \wedge [(X_4 \mid \{(\mathit{Seized}[\mathit{id}],\mathrm{false})\}) \vee (X_5 \mid \{(\mathit{Seized}[\mathit{id}-1],\mathrm{false})\})]\\
X_{11} &= X_8 \mid \{(C_{\mathrm{e},7}[\mathit{id}],0)\}\\
X_{12} &= \mathrm{false} \wedge X_2 \mid \{(C_{\mathrm{e},1}[\mathit{id}],0)\}\\
X_{\mathrm{x}} &= (C_{\mathrm{e},1}[\mathit{id}]=0) \wedge (C_{\mathrm{e},7}[\mathit{id}]=0) \wedge X_{\mathrm{e}} \wedge X_{11} \wedge X_{12}
\end{aligned}$$

After performing the insertions $6 \to 2$, $2 \to 9$, $3 \to 9$, $4 \to 10$, $5 \to 10$, $7 \to 8$, $\mathrm{e} \to 7$, $\mathrm{e} \to 1$, $8 \to 9$, and $8 \to 10$, we obtain the two following recursive equations for $X_9$ and $X_{10}$

$$\begin{aligned}
X_9 = {} & (X_{\mathrm{e}} \mid \{(E_9[\mathit{id}],0),(E_{10}[\mathit{id}],0)\} \vee X_9 \vee X_{10}) \wedge\\
& \{[(X_{\mathrm{e}} \vee ((E_{10}[\mathit{id}]=2) \wedge X_{10})) \wedge \neg \mathit{Seized}[\mathit{id}] \mid \{(E_9[\mathit{id}],1),(\mathit{Seized}[\mathit{id}],\mathrm{true})\}] \vee\\
& [((E_9[\mathit{id}]=1) \wedge X_9 \wedge \neg \mathit{Seized}[\mathit{id}-1]) \mid \{(E_9[\mathit{id}],2),(\mathit{Seized}[\mathit{id}-1],\mathrm{true})\}]\}
\end{aligned}$$

and

$$\begin{aligned}
X_{10} = {} & (X_{\mathrm{e}} \mid \{(E_9[\mathit{id}],0),(E_{10}[\mathit{id}],0)\} \vee X_9 \vee X_{10}) \wedge\\
& \{[(E_9[\mathit{id}]=2) \wedge X_9 \mid \{(E_{10}[\mathit{id}],1),(\mathit{Seized}[\mathit{id}],\mathrm{false})\}] \vee\\
& [(E_{10}[\mathit{id}]=1) \wedge X_{10} \mid \{(E_{10}[\mathit{id}],2),(\mathit{Seized}[\mathit{id}-1],\mathrm{false})\}]\}\,.
\end{aligned}$$

Simplifying the involved boolean expressions and solving the recurrence relations we obtain

$$X_8 = [X_\mathrm{e} \wedge \begin{cases} (E_9[id] = 1) \wedge \neg Seized[id-1] \mid \{(E_9[id], 2), (Seized[id-1], \mathrm{true})\}] \vee \\ (E_9[id] = 2) \mid \{(E_{10}[id], 1), (Seized[id], \mathrm{false})\}] \vee \\ (E_{10}[id] = 1) \mid \{(E_{10}[id], 2), (Seized[id-1], \mathrm{false})\}] \vee \\ (E_{10}[id] = 2) \wedge \neg Seized[id] \mid \{(E_9[id], 1), (Seized[id], \mathrm{true})\}] \end{cases}$$

Inserting e $\to$ 8, 8 $\to$ 11, 11 $\to$ x, 2 $\to$ 12, and 12 $\to$ x we see that $X_{12} = X_\mathrm{x} = \mathrm{false}$, which indicates that the program does not complete. On the other hand, this is clear because the philosopher tasks are supposed not to complete. It remains to check whether

$$\begin{aligned} X_8 = \bigwedge_{id} & [(E_9[id] = 1) \wedge \neg Seized[id-1] \mid \{(E_9[id], 2), (Seized[id-1], \mathrm{true})\}] \vee \\ & [(E_9[id] = 2) \mid \{(E_{10}[id], 1), (Seized[id], \mathrm{false})\}] \vee \\ & [(E_{10}[id] = 1) \mid \{(E_{10}[id], 2), (Seized[id-1], \mathrm{false})\}] \vee \\ & [(E_{10}[id] = 2) \wedge \neg Seized[id] \mid \{(E_9[id], 1), (Seized[id], \mathrm{true})\}] \end{aligned} \tag{1}$$

is true. Since,

$$\bigwedge_{id} [(E_{10}[id] = 2) \wedge \neg Seized[id]] = \mathrm{true} \tag{2}$$

implies

$$\bigwedge_{id} [(E_9[id] = 1) \wedge Seized[id]] = \bigwedge_{id} [(E_9[id] = 1) \wedge Seized[id-1]] = \mathrm{true},$$

we find that (1) evaluates to false if (2) holds. Thus we get

$$X_8 \Leftarrow \neg \bigwedge_{id} [(E_{10}[id] = 2) \wedge \neg Seized[id]] = \bigvee_{id} [(E_{10}[id] = 1) \vee Seized[id]]$$

which means that the program deadlocks if each philosopher holds exactly one fork and tries to pick up the second one, which is held by its neighbor. Thus our framework obtains the correct answer to the problem.

## 4 Related Work

A large number of papers deals with detecting tasking anomalies in multi-tasking (Ada) programs, e.g. [5, 7–11, 13, 18–20]. SSA form for explicitly parallel programs is treated theoretically in [18]. Explicitly parallel programs are restricted to a *(cobegin ... coend)*-like structure and are in strict contrast to the tasking in Ada. The semantic chosen for global variables is far too weak to model Ada programs with tasks. *Static Concurrency Analysis*, presented in [19], is a method for determining concurrency errors in parallel programs. The class of detectable errors include infinite waits, deadlocks, and concurrent updates of shared variables. Infeasible Paths, however, represent a problem since they give rise to errors which actually cannot occur. In [20] the authors propose symbolic execution to detect infeasible paths. Their approach is based on interleaving the execution of component processes. The interleaving approach, however, is poorly suited for formal verification. In [9, 10] concurrency analysis of programs is studied. The approach allows recursive procedures and dynamically allocated tasks to be present. The presented approach can handle (recursive) procedures but cannot take into account the individual "instances" of the procedures and tasks, i.e., it is based on the source-code of the tasks and procedures and not on their runtime equivalent. In [13–15] static detection of infinite wait anomalies is studied. Classic data flow analysis is employed to solve this problem. This allows polynomial time analysis but cannot solve dead paths problems. In addition, Ada's generic units cannot be modelled adequately. In [8] symbolic execution of Ada programs is presented. The analysis is restricted to a static task model where tasks can only be statically declared in the main program and the introduced framework cannot cope with dynamic task creation and completion. High-level Petri nets are employed in [5] to perform some analysis for multitasking Ada programs. In [11] Petri nets are used to reduce the state space of deadlock analysis for Ada programs. Petri nets can be used to perform a small class of deadlock detection, but they are not capable to analyze certain Ada features such as generic units. In [7] three different approaches to deadlock analysis are surveyed, namely reachability search, symbolic model checking, and inequality necessary conditions. None of the methods cited above can handle the Dining Philosophers problem where the number of philosophers enters the problem domain as a parameter. Our approach is different: it correctly reflects the runtime properties of multitasking (Ada) programs, it correctly models statically and dynamically allocated tasks, and it correctly handles generic units.

## 5 Conclusion and Future Work

We have presented a symbolic data flow analysis framework for detecting deadlocks. Our framework can handle dynamic task creation and completion, which goes beyond existing work. As shown in the dining philosophers example our framework can cope with generic units as well. Note that the task body is only analyzed once although several instances of the task can be either created statically or dynamically. Moreover, we observed that if the conditions like those of the dining philosophers example, are very complicated, the program has good chances to deadlock. Deadlock-free tasking programs usually lead to simple conditions.

Our approach is also well-suited for other programming languages. In future we plan to build a similar analysis for object-oriented programming languages, in particular for Java.

## References

1. J. Blieberger, *Data-flow frameworks for worst-case execution time analysis*, Real-Time Systems (2000), (to appear).

2. J. Blieberger and B. Burgstaller, *Symbolic reaching definitions analysis of Ada programs*, Proceedings of Ada-Europe'98 (Uppsala, Sweden), pp. 238–250.
3. J. Blieberger, B. Burgstaller, and B. Scholz, *Interprocedural Symbolic Evaluation of Ada Programs with Aliases*, Ada-Europe'99 International Conference on Reliable Software Technologies (Santander, Spain), pp. 136–145.
4. J. Blieberger, T. Fahringer, and B. Scholz, *Symbolic cache analysis for real-time systems*, Real-Time Systems, Special Issue on Worst-Case Execution Time Analysis (2000), (to appear).
5. E. Bruneton and J.-F. Pradat-Peyre, *Automatic verification of concurrent Ada programs*, Ada-Europe'99 International Conference on Reliable Software Technologies (Santander, Spain), pp. 146–157.
6. T. E. Cheatham, G. H. Holloway, and J. A. Townley, *Symbolic evaluation and the analysis of programs*, IEEE Trans. on Software Engineering **5** (1979), no. 4, 403–417.
7. J. C. Corbett, *Evaluating deadlock detection methods for concurrent software*, IEEE Transactions on Software Engineering **22** (1996), no. 3, 161–180.
8. L. K. Dillon, *Using symbolic execution for verification of Ada tasking programs*, ACM Transactions on Programming Languages and Systems **12** (1990), no. 4, 643–669.
9. E. Duesterwald, *Static concurrency analysis in the presence of procedures*, Tech. Report #91-6, Department of Computer Science, University of Pittsburgh, 1991.
10. E. Duesterwald and M. L. Soffa, *Concurrency analysis in the presence of procedures using a data-flow framework*, Proceedings of the 4th Symp. on Testing, Analysis and Verification (TAV4), pp. 36–48.
11. S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz, *Application and experimental evaluation of state space reduction methods for deadlock analysis in Ada*, ACM Trans. on Software Engineering and Methodology **3** (1994), no. 4, 161–180.
12. T. Fahringer and B. Scholz, *Symbolic Evaluation for Parallelizing Compilers*, Proc. of the ACM International Conference on Supercomputing.
13. D. Long and L. A. Clarke, *Data flow analysis of concurrent systems that use the rendezvous model of synchronization*, Proceedings of the ACM Symp. on Testing, Analysis, and Verification, pp. 21–35.
14. S. P. Masticola, *Static detection of deadlocks in polynomial time*, Ph.D. thesis, Graduate School—New Brunswick, Rutgers, The State University of New Jersey, 1993.
15. S. P. Masticola and B. G. Ryder, *Static infinite wait anomaly detection in polynomial time*, Proceedings of the 1990 International Conference on Parallel Processing, pp. II78–II87.
16. B. G. Ryder and M. C. Paull, *Elimination algorithms for data flow analysis*, ACM Computing Surveys **18** (1986), no. 3, 277–315.
17. B. Scholz, J. Blieberger, and T. Fahringer, *Symbolic Pointer Analysis for Detecting Memory Leaks*, ACM SIGPLAN Workshop on "Partial Evaluation and Semantics-Based Program Manipulation" (PEPM'00) (Boston).
18. E. Stoltz, H. Srinivasan, J. Hook, and M. Wolfe, *Static single assignment form for explicitly parallel programs: Theory and practice*, Tech. report, Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, Portland, Oregon, 1994.
19. R. N. Taylor, *A general-purpose algorithm for analyzing concurrent programs*, Communications of the ACM **26** (1983), no. 5, 362–376.
20. M. Young and R. N. Taylor, *Combining static concurrency analysis with symbolic execution*, IEEE Trans. on Software Engineering **14** (1988), no. 10, 1499–1511.