Kronecker Algebra for Static Analysis of Barriers in Ada

Robert Mittermayr, Johann Blieberger

Institute of Computer Aided Automation
TU Vienna, Austria
{robert,blieb}@auto.tuwien.ac.at

June 16th, 2016

Outline

1. Preliminaries and Modelling

Outline

Outline

Outline

Outline

Preliminaries and Modelling

- tasks and synchronization primitives represented by control flow graphs (CFGs)

Preliminaries and Modelling

- tasks and synchronization primitives represented by control flow graphs (CFGs)
- CFG stored in form of adjacency matrix

Preliminaries and Modelling

- tasks and synchronization primitives represented by control flow graphs (CFGs)
- CFG stored in form of adjacency matrix
- CFG edges labeled by elements of a semiring (compare: automata, DFAs, regular expressions)

Preliminaries and Modelling

- tasks and synchronization primitives represented by control flow graphs (CFGs)
- CFG stored in form of adjacency matrix
- CFG edges labeled by elements of a semiring (compare: automata, DFAs, regular expressions)
- set of labels $\mathcal{L} = \mathcal{L}_\mathsf{V} \cup \mathcal{L}_\mathsf{S}$, where

Preliminaries and Modelling

- tasks and synchronization primitives represented by control flow graphs (CFGs)
- CFG stored in form of adjacency matrix
- CFG edges labeled by elements of a semiring (compare: automata, DFAs, regular expressions)
- set of labels $\mathcal{L} = \mathcal{L}_\mathsf{V} \cup \mathcal{L}_\mathsf{S}$, where
  - $\mathcal{L}_\mathsf{V}$ . . . set of non-synchronization labels and

Preliminaries and Modelling

- tasks and synchronization primitives represented by control flow graphs (CFGs)
- CFG stored in form of adjacency matrix
- CFG edges labeled by elements of a semiring (compare: automata, DFAs, regular expressions)
- set of labels $\mathcal{L} = \mathcal{L}_\mathsf{V} \cup \mathcal{L}_\mathsf{S}$, where
  - $\mathcal{L}_\mathsf{V}$ ... set of non-synchronization labels and
  - $\mathcal{L}_\mathsf{S}$ ... set of labels representing calls to synchronization primitives

Preliminaries and Modelling

- tasks and synchronization primitives represented by control flow graphs (CFGs)
- CFG stored in form of adjacency matrix
- CFG edges labeled by elements of a semiring (compare: automata, DFAs, regular expressions)
- set of labels $\mathcal{L} = \mathcal{L}_V \cup \mathcal{L}_S$, where
  - $\mathcal{L}_V$ ... set of non-synchronization labels and
  - $\mathcal{L}_S$ ... set of labels representing calls to synchronization primitives
  - $\mathcal{L}_V$ and $\mathcal{L}_S$ are disjoint

Preliminaries and Modelling

- tasks and synchronization primitives represented by control flow graphs (CFGs)
- CFG stored in form of adjacency matrix
- CFG edges labeled by elements of a semiring (compare: automata, DFAs, regular expressions)
- set of labels $\mathcal{L} = \mathcal{L}_\mathsf{V} \cup \mathcal{L}_\mathsf{S}$, where
  - $\mathcal{L}_\mathsf{V}$ ... set of non-synchronization labels and
  - $\mathcal{L}_\mathsf{S}$ ... set of labels representing calls to synchronization primitives
  - $\mathcal{L}_\mathsf{V}$ and $\mathcal{L}_\mathsf{S}$ are disjoint
- matrices out of $\mathcal{M} = \{M = (m_{i,j}) \,|\, m_{i,j} \in \mathcal{L}\}$ only.

Preliminaries and Modelling

- Edge splitting: for synchronization primitive calls (e.g. *p*- and *v*-calls to semaphores); the only statement on the corresponding (split) edge.

Preliminaries and Modelling

- Edge splitting: for synchronization primitive calls (e.g. *p*- and *v*-calls to semaphores); the only statement on the corresponding (split) edge.
- system model . . . tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where

Preliminaries and Modelling

- Edge splitting: for synchronization primitive calls (e.g. *p*- and *v*-calls to semaphores); the only statement on the corresponding (split) edge.
- system model ... tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where
  - $\mathcal{T}$ ... finite set of CFG adjacency matrices describing tasks,

Preliminaries and Modelling

- Edge splitting: for synchronization primitive calls (e.g. *p*- and *v*-calls to semaphores); the only statement on the corresponding (split) edge.
- system model . . . tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where
  - $\mathcal{T}$ . . . finite set of CFG adjacency matrices describing tasks,
  - $\mathcal{S}$ . . . finite set of CFG adjacency matrices describing synchronization primitives (e.g. semaphores), and

Preliminaries and Modelling

- Edge splitting: for synchronization primitive calls (e.g. p- and v-calls to semaphores); the only statement on the corresponding (split) edge.
- system model ... tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where
  - $\mathcal{T}$ ... finite set of CFG adjacency matrices describing tasks,
  - $\mathcal{S}$ ... finite set of CFG adjacency matrices describing synchronization primitives (e.g. semaphores), and
  - labels in $T \in \mathcal{T}$ and $S \in \mathcal{S}$ ... elements of $\mathcal{L}$ and $\mathcal{L}_S$, respectively.

Preliminaries and Modelling

- Edge splitting: for synchronization primitive calls (e.g. *p*- and *v*-calls to semaphores); the only statement on the corresponding (split) edge.
- system model ... tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where
  - $\mathcal{T}$ ... finite set of CFG adjacency matrices describing tasks,
  - $\mathcal{S}$ ... finite set of CFG adjacency matrices describing synchronization primitives (e.g. semaphores), and
  - labels in $T \in \mathcal{T}$ and $S \in \mathcal{S}$ ... elements of $\mathcal{L}$ and $\mathcal{L}_S$, respectively.
- *Concurrent Program Graph* (CPG) ... a graph $C = \langle V, E, n_e \rangle$ with

Preliminaries and Modelling

- Edge splitting: for synchronization primitive calls (e.g. p- and v-calls to semaphores); the only statement on the corresponding (split) edge.
- system model ... tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where
  - $\mathcal{T}$ ... finite set of CFG adjacency matrices describing tasks,
  - $\mathcal{S}$ ... finite set of CFG adjacency matrices describing synchronization primitives (e.g. semaphores), and
  - labels in $T \in \mathcal{T}$ and $S \in \mathcal{S}$ ... elements of $\mathcal{L}$ and $\mathcal{L}_\mathsf{S}$, respectively.
- *Concurrent Program Graph* (CPG) ... a graph $C = \langle V, E, n_e \rangle$ with
  - set of nodes V,

Preliminaries and Modelling

- Edge splitting: for synchronization primitive calls (e.g. p- and v-calls to semaphores); the only statement on the corresponding (split) edge.
- system model ... tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where
  - $\mathcal{T}$ ... finite set of CFG adjacency matrices describing tasks,
  - $\mathcal{S}$ ... finite set of CFG adjacency matrices describing synchronization primitives (e.g. semaphores), and
  - labels in $T \in \mathcal{T}$ and $S \in \mathcal{S}$ ... elements of $\mathcal{L}$ and $\mathcal{L}_S$, respectively.
- *Concurrent Program Graph* (CPG) ... a graph $C = \langle V, E, n_e \rangle$ with
  - set of nodes V,
  - set of directed edges $E \subseteq V \times V$, and

Preliminaries and Modelling

- Edge splitting: for synchronization primitive calls (e.g. p- and v-calls to semaphores); the only statement on the corresponding (split) edge.
- system model . . . tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where
  - $\mathcal{T}$ . . . finite set of CFG adjacency matrices describing tasks,
  - $\mathcal{S}$ . . . finite set of CFG adjacency matrices describing synchronization primitives (e.g. semaphores), and
  - labels in $T \in \mathcal{T}$ and $S \in \mathcal{S}$ . . . elements of $\mathcal{L}$ and $\mathcal{L}_S$, respectively.
- *Concurrent Program Graph* (CPG) . . . a graph $C = \langle V, E, n_e \rangle$ with
  - set of nodes V,
  - set of directed edges $E \subseteq V \times V$, and
  - so-called *entry* node $n_e \in V$

Preliminaries and Modelling

- Edge splitting: for synchronization primitive calls (e.g. p- and v-calls to semaphores); the only statement on the corresponding (split) edge.
- system model ... tuple $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$, where
  - $\mathcal{T}$ ... finite set of CFG adjacency matrices describing tasks,
  - $\mathcal{S}$ ... finite set of CFG adjacency matrices describing synchronization primitives (e.g. semaphores), and
  - labels in $T \in \mathcal{T}$ and $S \in \mathcal{S}$ ... elements of $\mathcal{L}$ and $\mathcal{L}_S$, respectively.
- *Concurrent Program Graph* (CPG) ... a graph $C = \langle V, E, n_e \rangle$ with
  - set of nodes V,
  - set of directed edges $E \subseteq V \times V$, and
  - so-called *entry* node $n_e \in V$
- sets V and E constructed out of the elements of $\langle \mathcal{T}, \mathcal{S}, \mathcal{L} \rangle$.

Kronecker Product

Given an m-by-n matrix A and a p-by-q matrix B, their *Kronecker product* $A \otimes B$ is an mp-by-nq block matrix defined by

$$A \otimes B = \begin{pmatrix} a_{1,1} \cdot B & \cdots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot B & \cdots & a_{m,n} \cdot B \end{pmatrix}.$$

Kronecker Product

Given an m-by-n matrix A and a p-by-q matrix B, their *Kronecker product* $A \otimes B$ is an mp-by-nq block matrix defined by

$$A \otimes B = \begin{pmatrix} a_{1,1} \cdot B & \cdots & a_{1,n} \cdot B \\ \vdots & \ddots & \vdots \\ a_{m,1} \cdot B & \cdots & a_{m,n} \cdot B \end{pmatrix}.$$

Given two automata, the Kronecker product synchronously executes them (lock-step).

Kronecker Sum

Given a matrix A of order m and a matrix B of order n, their *Kronecker sum* $A \oplus B$ is a matrix of order mn defined by

$$A \oplus B = A \otimes I_n + I_m \otimes B$$

where I_m and I_n denote identity matrices of order m and n, respectively.

Kronecker Sum

Given a matrix A of order m and a matrix B of order n, their *Kronecker sum* $A \oplus B$ is a matrix of order mn defined by

$$A \oplus B = A \otimes I_n + I_m \otimes B$$

where I_m and I_n denote identity matrices of order m and n, respectively.

- Kronecker sum calculates all possible interleavings of two concurrently executing automata

Kronecker Sum

Given a matrix A of order m and a matrix B of order n, their *Kronecker sum* $A \oplus B$ is a matrix of order mn defined by

$$A \oplus B = A \otimes I_n + I_m \otimes B$$

where I_m and I_n denote identity matrices of order m and n, respectively.

- Kronecker sum calculates all possible interleavings of two concurrently executing automata
- even if the automata contain conditionals and loops.

Selective Kronecker Product

$\oslash_L$ limits synchronization of the operands to labels $l \in L \subseteq \mathcal{L}$.

Selective Kronecker Product

$\oslash_L$ limits synchronization of the operands to labels $l \in L \subseteq \mathcal{L}$.

Given an m-by-n matrix A and a p-by-q matrix B, we call $A \oslash_L B$ their selective Kronecker product. For all $l \in L \subseteq \mathcal{L}$ let
$A \oslash_L B = (a_{i,j}) \oslash_L (b_{r,s}) = (c_{t,u})$, where

$$c_{(i-1)\cdot p+r,(j-1)\cdot q+s} = \begin{cases} l & \text{if } a_{i,j} = b_{r,s} = l,\ l \in L, \\ 0 & \text{otherwise.} \end{cases}$$

Selective Kronecker Product

$\oslash_L$ limits synchronization of the operands to labels $l \in L \subseteq \mathcal{L}$.

Given an m-by-n matrix A and a p-by-q matrix B, we call $A \oslash_L B$ their selective Kronecker product. For all $l \in L \subseteq \mathcal{L}$ let
$A \oslash_L B = (a_{i,j}) \oslash_L (b_{r,s}) = (c_{t,u})$, where

$$c_{(i-1)\cdot p+r,(j-1)\cdot q+s} = \begin{cases} l & \text{if } a_{i,j} = b_{r,s} = l, \ l \in L, \\ 0 & \text{otherwise.} \end{cases}$$

Selective Kronecker product ensures that, e.g., a semaphore p-call in the left operand is paired with the p-operation in the right operand and not with any other operation in the right operand. In practice, we usually constrain $L \subseteq \mathcal{L}_S$.

Filtered Matrix

We call M_L a *filtered matrix* and define it as a matrix of order $o(M)$ containing entries of $L \subseteq \mathcal{L}$ of $M = (m_{i,j})$ and zeros elsewhere:

$$M_L = (m_{L;i,j}), \text{ where } m_{L;i,j} = \begin{cases} m_{i,j} & \text{if } m_{i,j} \in L, \\ 0 & \text{otherwise.} \end{cases}$$

System Model

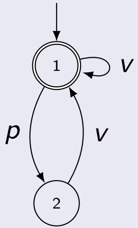
The adjacency matrix representing program $\mathcal{P}$ is referred to as P (= CPG).

System Model

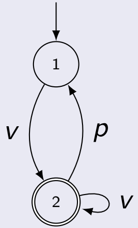
The adjacency matrix representing program $\mathcal{P}$ is referred to as P (= CPG). P can be efficiently computed by

$$P = T \oslash_{\mathcal{L}_\mathrm{S}} S + T_{\mathcal{L}_\mathrm{V}} \otimes I_{o(S)}.$$

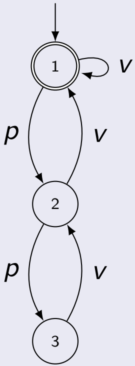
Examples



Initially Unlocked Binary Semaphore

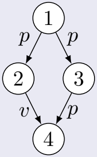


Initially Locked Binary Semaphore



Counting Semaphore

Examples

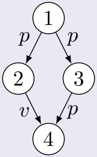


(a) CFG

$$\begin{pmatrix} 0 & p & p & 0 \\ 0 & 0 & 0 & v \\ 0 & 0 & 0 & p \\ 0 & 0 & 0 & 0 \end{pmatrix} \oslash_{\{p,v\}} \begin{pmatrix} v & p \\ v & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & p & 0 & p & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & p \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(b) Matrices

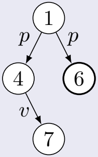
Examples



(a) CFG

$$\begin{pmatrix} 0 & p & p & 0 \\ 0 & 0 & 0 & v \\ 0 & 0 & 0 & p \\ 0 & 0 & 0 & 0 \end{pmatrix} \oslash_{\{p,v\}} \begin{pmatrix} v & p \\ v & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 0 & p & 0 & p & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & v & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & p \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(b) Matrices



self-deadlock at node 6

Barriers

- Barrier is a synchronization construct available in most modern programming languages (e.g. Ada and Java).

Barriers

- Barrier is a synchronization construct available in most modern programming languages (e.g. Ada and Java).
- Barrier is used to synchronize a set of n threads.

Barriers

- Barrier is a synchronization construct available in most modern programming languages (e.g. Ada and Java).
- Barrier is used to synchronize a set of n threads.
- The first thread(s) reaching the barrier will be blocked.

Barriers

- Barrier is a synchronization construct available in most modern programming languages (e.g. Ada and Java).
- Barrier is used to synchronize a set of n threads.
- The first thread(s) reaching the barrier will be blocked.
- When the nth thread reaches the barrier, all the threads are released and continue their work.

Barriers

- Barrier is a synchronization construct available in most modern programming languages (e.g. Ada and Java).
- Barrier is used to synchronize a set of *n* threads.
- The first thread(s) reaching the barrier will be blocked.
- When the *n*th thread reaches the barrier, all the threads are released and continue their work.
- Barrier is called *reusable*, when it can be re-used after the threads are released.

Barriers

- Barrier is a synchronization construct available in most modern programming languages (e.g. Ada and Java).
- Barrier is used to synchronize a set of n threads.
- The first thread(s) reaching the barrier will be blocked.
- When the nth thread reaches the barrier, all the threads are released and continue their work.
- Barrier is called *reusable*, when it can be re-used after the threads are released.
- *Static barriers* have a statically fixed number of participating tasks/threads.

Barriers

- Barrier is a synchronization construct available in most modern programming languages (e.g. Ada and Java).
- Barrier is used to synchronize a set of n threads.
- The first thread(s) reaching the barrier will be blocked.
- When the nth thread reaches the barrier, all the threads are released and continue their work.
- Barrier is called *reusable*, when it can be re-used after the threads are released.
- *Static barriers* have a statically fixed number of participating tasks/threads.
- The number of threads can vary at runtime for *dynamic barriers*.

Barriers in Ada (Annex D)

```
package Ada.Synchronous_Barriers is
    pragma Preelaborate(Synchronous_Barriers);
    subtype Barrier_Limit is Positive range 1 .. implementation-defined;
    type Synchronous_Barrier (Release_Threshold : Barrier_Limit) is limited private;
    procedure Wait_For_Release (The_Barrier : in out Synchronous_Barrier;
        Notified : out Boolean);
private
    -- not specified by the language
end Ada.Synchronous_Barriers;
```

Barriers in Ada (Annex D)

```
package Ada.Synchronous_Barriers is
    pragma Preelaborate(Synchronous_Barriers);
    subtype Barrier_Limit is Positive range 1 .. implementation-defined;
    type Synchronous_Barrier (Release_Threshold : Barrier_Limit) is limited private;
    procedure Wait_For_Release (The_Barrier : in out Synchronous_Barrier;
        Notified : out Boolean);
private
    -- not specified by the language
end Ada.Synchronous_Barriers;
```

Ada's barriers are static and reusable

Barriers in Ada (Annex D)

```
package Ada.Synchronous_Barriers is
    pragma Preelaborate(Synchronous_Barriers);
    subtype Barrier_Limit is Positive range 1 .. implementation-defined;
    type Synchronous_Barrier (Release_Threshold : Barrier_Limit) is limited private;
    procedure Wait_For_Release (The_Barrier : in out Synchronous_Barrier;
        Notified : out Boolean);
private
    -- not specified by the language
end Ada.Synchronous_Barriers;
```

Ada's barriers are static and reusable

Java has static and dynamic barriers (reusable and non-reuseable)

Barriers in Ada (Annex D)

```
package Ada.Synchronous_Barriers is
    pragma Preelaborate(Synchronous_Barriers);
    subtype Barrier_Limit is Positive range 1 .. implementation-defined;
    type Synchronous_Barrier (Release_Threshold : Barrier_Limit) is limited private;
    procedure Wait_For_Release (The_Barrier : in out Synchronous_Barrier;
        Notified : out Boolean);
private
    -- not specified by the language
end Ada.Synchronous_Barriers;
```

Ada's barriers are static and reusable

Java has static and dynamic barriers (reusable and non-reuseable)

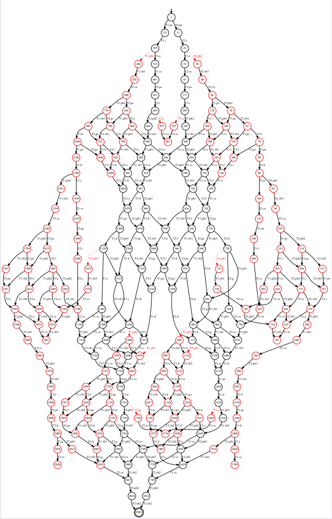
In this paper: only static barriers

Implementation of Barriers

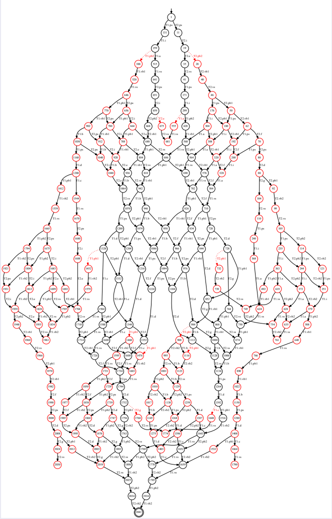
Reusable Barrier Solution using Semaphores

```
mutex.wait()                 # ps
    count += 1               # i
    if count == n:
        turnstile2.wait()    # pb2, lock the second
        turnstile.signal()   # vb1, unlock the first
    else # empty             # T1.a; T2.e
mutex.signal()               # vs

turnstile.wait()             # pb1, first turnstile
turnstile.signal()           # vb1

# critical point             # T1.b; T2.f

mutex.wait()                 # ps
    count -= 1               # d
    if count == 0:
        turnstile.wait()     # pb1, lock the first
        turnstile2.signal()  # vb2, unlock the second
    else # empty             # T1.c; T2.g
mutex.signal()               # vs

turnstile2.wait()            # pb2, second turnstile
turnstile2.signal()          # vb2
```

Reusable Barrier Solution using Semaphores

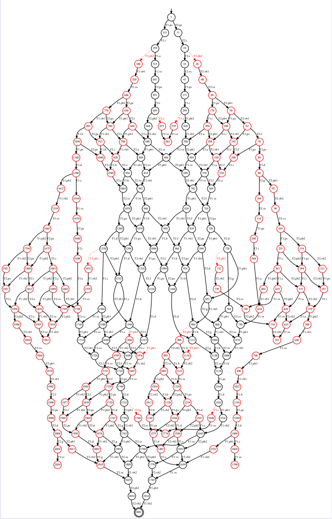


Reusable Barrier Solution using Semaphores



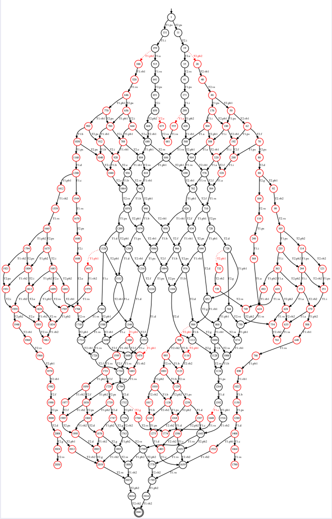
- The CPG contains *potential* deadlock nodes 681, 761, 1774, 1790, 1961 and 2030.

Reusable Barrier Solution using Semaphores



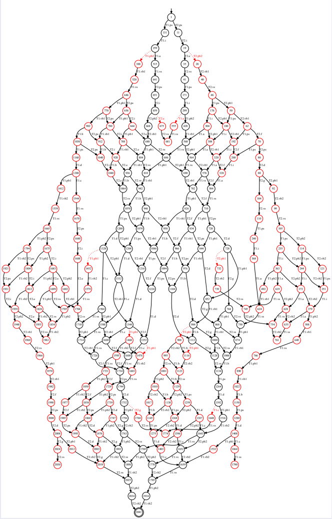
- The CPG contains *potential* deadlock nodes 681, 761, 1774, 1790, 1961 and 2030.
- The dotted edges are dead paths which can be ruled out by a value-sensitive (e.g. symbolic) analysis.

Reusable Barrier Solution using Semaphores



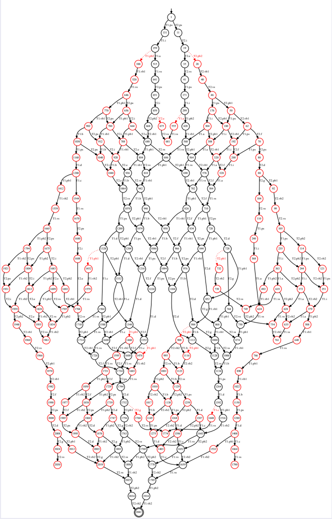
- The CPG contains *potential* deadlock nodes 681, 761, 1774, 1790, 1961 and 2030.
- The dotted edges are dead paths which can be ruled out by a value-sensitive (e.g. symbolic) analysis.
- Due to these edges some nodes are unreachable which are colored in red.

Reusable Barrier Solution using Semaphores



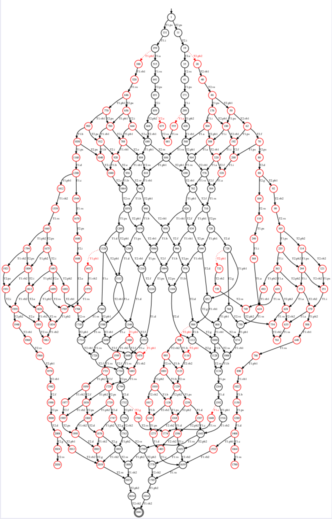
- The CPG contains *potential* deadlock nodes 681, 761, 1774, 1790, 1961 and 2030.
- The dotted edges are dead paths which can be ruled out by a value-sensitive (e.g. symbolic) analysis.
- Due to these edges some nodes are unreachable which are colored in red.
- All potential deadlock nodes are unreachable.

Reusable Barrier Solution using Semaphores



- The CPG contains *potential* deadlock nodes 681, 761, 1774, 1790, 1961 and 2030.
- The dotted edges are dead paths which can be ruled out by a value-sensitive (e.g. symbolic) analysis.
- Due to these edges some nodes are unreachable which are colored in red.
- All potential deadlock nodes are unreachable.
- Implementation using three semaphores is correct.

Reusable Barrier Solution using Semaphores



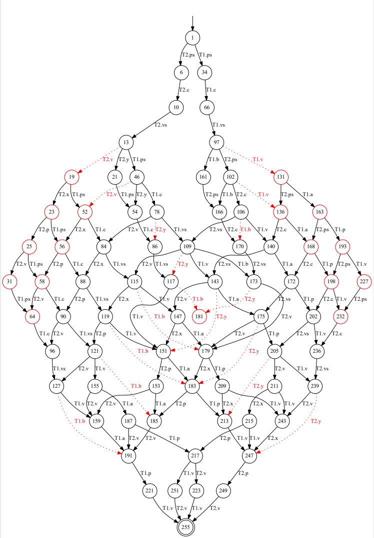
- The CPG contains *potential* deadlock nodes 681, 761, 1774, 1790, 1961 and 2030.
- The dotted edges are dead paths which can be ruled out by a value-sensitive (e.g. symbolic) analysis.
- Due to these edges some nodes are unreachable which are colored in red.
- All potential deadlock nodes are unreachable.
- Implementation using three semaphores is correct.
- Advanced approaches like symbolic analysis are needed.

Implementation of Barriers

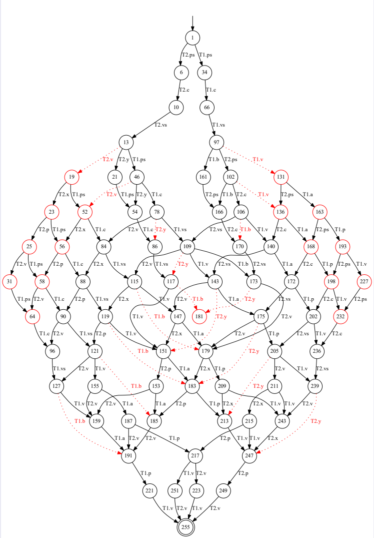
Non-Reusable Barrier Solution using Semaphores

```
# rendezvous

mutex.wait()                       # ps
    count = count + 1              # c
mutex.signal()                     # vs

if count == n: barrier.signal()    # T1.v, T1.a; T2.v, T2.x
else # empty                       # T1.b; T2.y

barrier.wait()                     # p
barrier.signal()                   # v

# critical point
```

Non-Reusable Barrier Solution using Semaphores

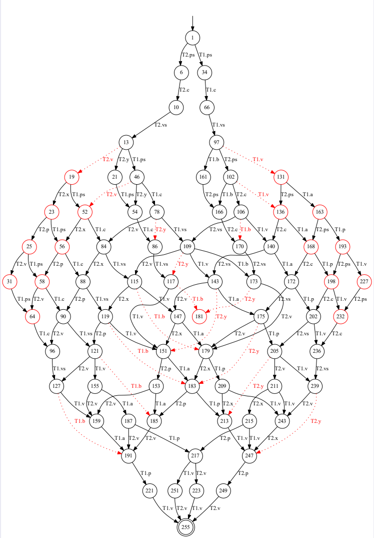


Non-Reusable Barrier Solution using Semaphores



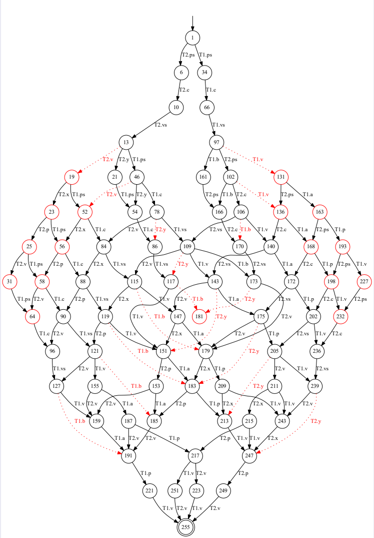
- Again there are dead paths (the corresponding edges are dotted).

Non-Reusable Barrier Solution using Semaphores



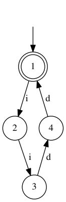
- Again there are dead paths (the corresponding edges are dotted).
- Deadlock node (node 181).

Non-Reusable Barrier Solution using Semaphores

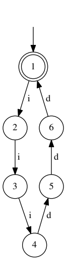


- Again there are dead paths (the corresponding edges are dotted).
- Deadlock node (node 181).
- Paths to node 181 are dead paths.

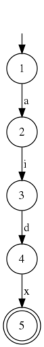
Barrier Synchronization Object



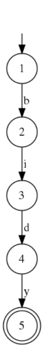
size 2
barrier object



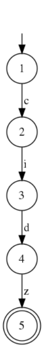
size 3
barrier object



user
thread 1

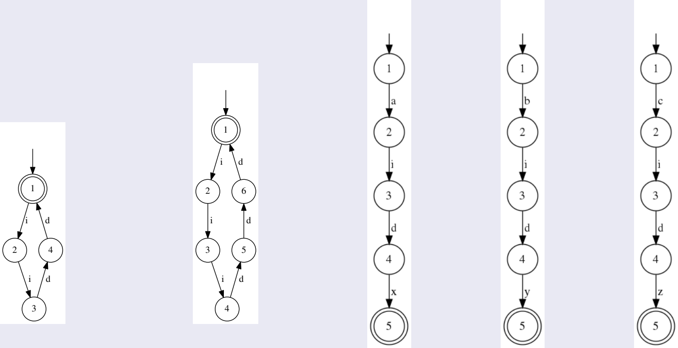


user
thread 2



user
thread 3

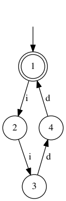
Barrier Synchronization Object



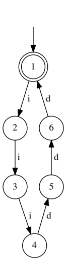
size 2 barrier object | size 3 barrier object | user thread 1 | user thread 2 | user thread 3

- models "semantics of barriers" instead of "implementation"

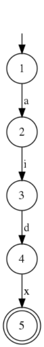
Barrier Synchronization Object



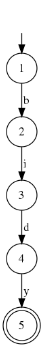
size 2
barrier object



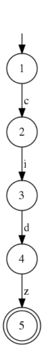
size 3
barrier object



user
thread 1



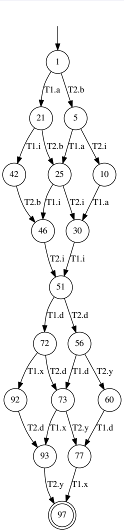
user
thread 2



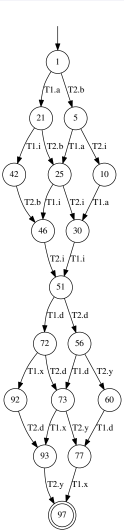
user
thread 3

- models “semantics of barriers” instead of “implementation”
- cannot verify implementation

Barrier Synchronization Object – Example: Two threads

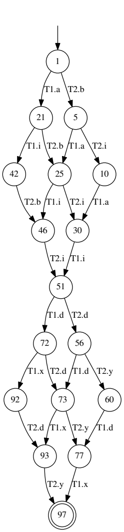


Barrier Synchronization Object – Example: Two threads



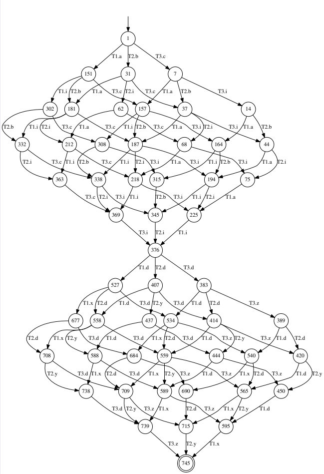
- free of deadlocks

Barrier Synchronization Object – Example: Two threads

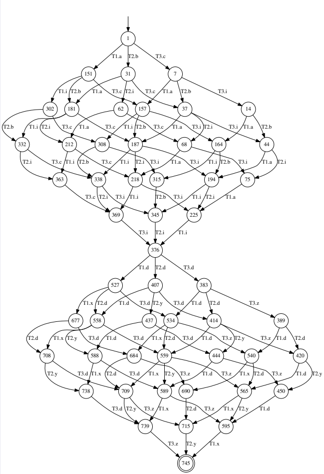


- free of deadlocks
- no need for value sensitive analysis

Barrier Synchronization Object – Example: Three threads

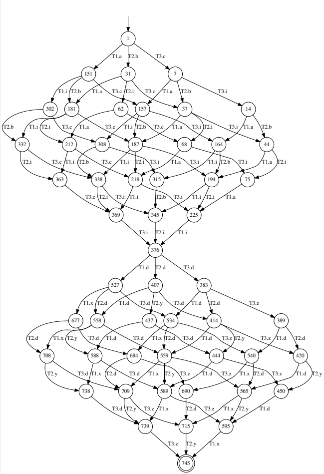


Barrier Synchronization Object – Example: Three threads



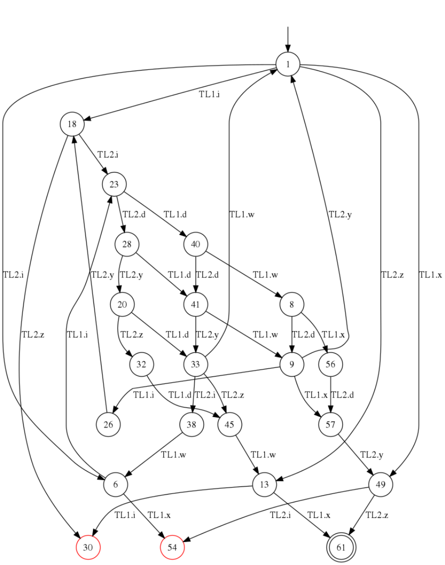
- free of deadlocks

Barrier Synchronization Object – Example: Three threads



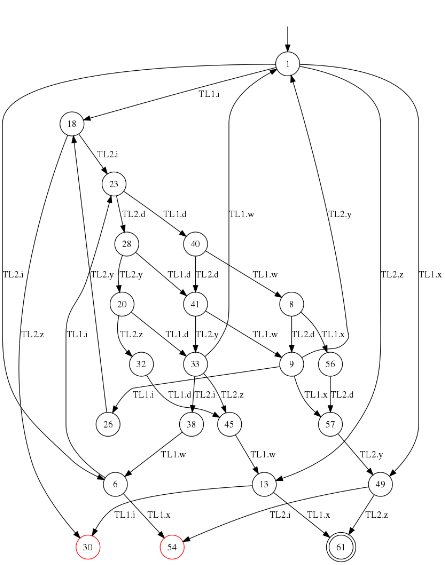
- free of deadlocks
- no need for value sensitive analysis

Barrier Synchronization Object – Example with Loops



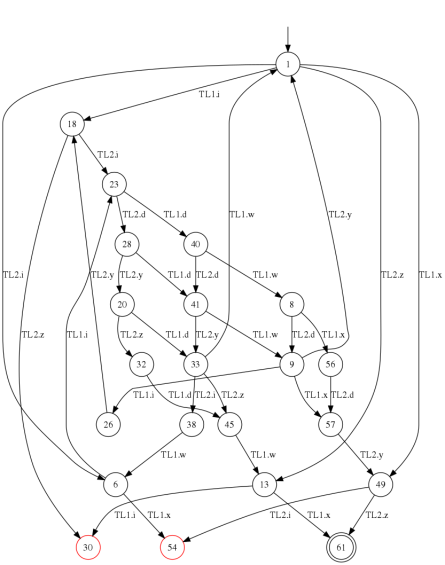
- Each task contains a loop and a `Wait_For_Release` inside the loop.

Barrier Synchronization Object – Example with Loops



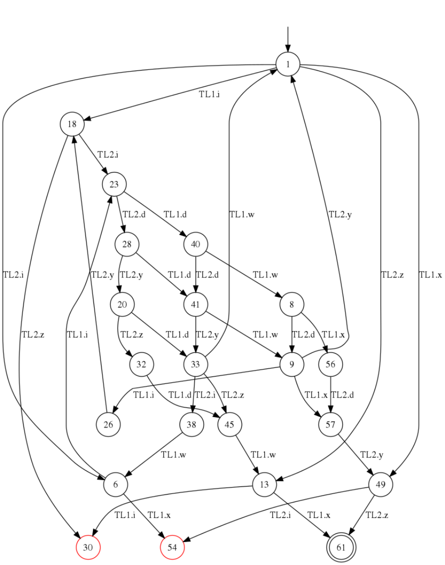
- Each task contains a loop and a `Wait_For_Release` inside the loop.
- If the number of loop iterations is the same in both tasks, the final node 61 is reached; otherwise, the program stalls at nodes 30 or 54.

Barrier Synchronization Object – Example with Loops



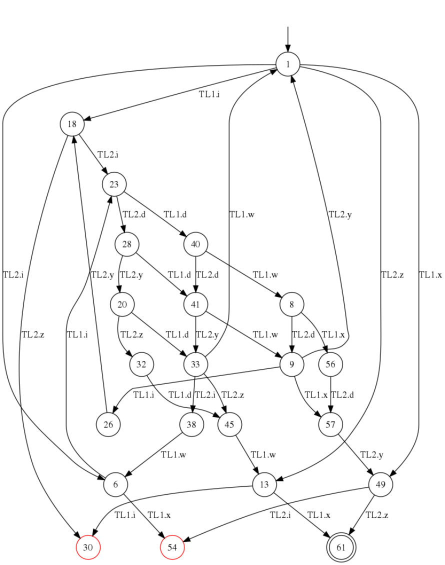
- Each task contains a loop and a `Wait_For_Release` inside the loop.
- If the number of loop iterations is the same in both tasks, the final node 61 is reached; otherwise, the program stalls at nodes 30 or 54.
- The number of loop iterations cannot be calculated by the Kronecker approach.

Barrier Synchronization Object – Example with Loops



- Each task contains a loop and a `Wait_For_Release` inside the loop.
- If the number of loop iterations is the same in both tasks, the final node 61 is reached; otherwise, the program stalls at nodes 30 or 54.
- The number of loop iterations cannot be calculated by the Kronecker approach.
- For this purpose e.g. some sort of symbolic analysis is needed.

Barrier Synchronization Object – Example with Loops



- Each task contains a loop and a `Wait_For_Release` inside the loop.
- If the number of loop iterations is the same in both tasks, the final node 61 is reached; otherwise, the program stalls at nodes 30 or 54.
- The number of loop iterations cannot be calculated by the Kronecker approach.
- For this purpose e.g. some sort of symbolic analysis is needed.
- In the simplest case, only lower and upper bounds of for-loops have to be compared.

Conclusions

- Kronecker algebra for static analysis of concurrent Ada programs with reusable static barriers for synchronization.

Conclusions

- Kronecker algebra for static analysis of concurrent Ada programs with reusable static barriers for synchronization.
- Compared our novel barrier synchronization primitive with a barrier implementation based on semaphores.

Conclusions

- Kronecker algebra for static analysis of concurrent Ada programs with reusable static barriers for synchronization.
- Compared our novel barrier synchronization primitive with a barrier implementation based on semaphores.
- Implementations using semaphores require advanced techniques to find dead paths.

Conclusions

- Kronecker algebra for static analysis of concurrent Ada programs with reusable static barriers for synchronization.
- Compared our novel barrier synchronization primitive with a barrier implementation based on semaphores.
- Implementations using semaphores require advanced techniques to find dead paths.
- Our barrier construct can be analyzed by static analysis only.

Conclusions

- Kronecker algebra for static analysis of concurrent Ada programs with reusable static barriers for synchronization.
- Compared our novel barrier synchronization primitive with a barrier implementation based on semaphores.
- Implementations using semaphores require advanced techniques to find dead paths.
- Our barrier construct can be analyzed by static analysis only.
- Need advanced techniques for programs containing loops or conditional statements.