The Complexity of Bounded Context Switching with Dynamic Thread Creation

Pascal Baumann, Rupak Majumdar, Ramanathan S. Thinniyam, Georg Zetzsche

Max Planck Institute for Software Systems (MPI-SWS)

MOVEP 2020 (originally ICALP 2020)

Introduction

Example Program

```
global lock l;

main() {
  spawn handler();
  spawn main();
}

handler() {
  if * {
    a(0);
  } else {
    a(1);
  }
  unlock(l);
}
```

```
a(i) {
  if * {
    a(0);
  } elsif * {
    a(1);
  } else {
    lock(l);
  }

  // critical section:
  write(i);

  return;
}
```

Model Features

To model the behavior of the example, we need:

- Concurrent threads spawning dynamically during execution.
- A finite global memory, accessable by all threads.
- A local stack for each thread (e.g. to store procedure calls).
- Bound K on the number of context switches avoids undecidability.

Model Features

To model the behavior of the example, we need:

- Concurrent threads spawning dynamically during execution.
- A finite global memory, accessable by all threads.
- A local stack for each thread (e.g. to store procedure calls).
- Bound K on the number of context switches avoids undecidability.

For $K = 0$ safety is EXPSPACE-complete.

- Shown by Ganty and Majumdar (2012).

For $K \geq 1$ safety is EXPSPACE-hard and in 2EXPSPACE.

- Shown by Atig, Bouajjani, and Qadeer (2009).

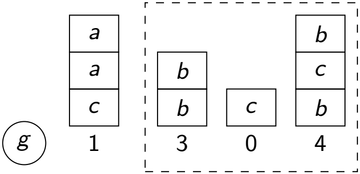
Our result closes the remaining complexity gap.

Dynamic Networks of Concurrent Pushdown Systems (DCPS)

Configurations

Consist of:

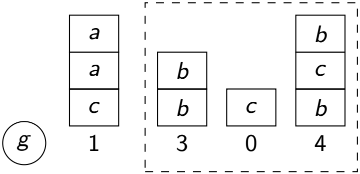
- A global state,
- an active thread with
  - a local stack
  - and a context switch number,
- and a bag of inactive threads.



Configurations

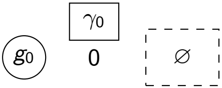
Consist of:

- A global state,
- an active thread with
  - a local stack
  - and a context switch number,
- and a bag of inactive threads.



Initial configuration:

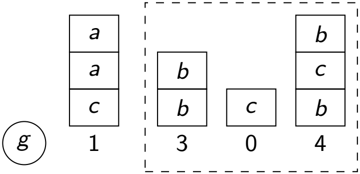
- Initial state g_0,
- an active thread with
  - only the initial symbol γ_0 on the stack
  - and context switch number 0,
- and an empty bag.



Configurations

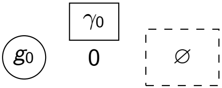
Consist of:

- A global state,
- an active thread with
  - a local stack
  - and a context switch number,
- and a bag of inactive threads.



Initial configuration:

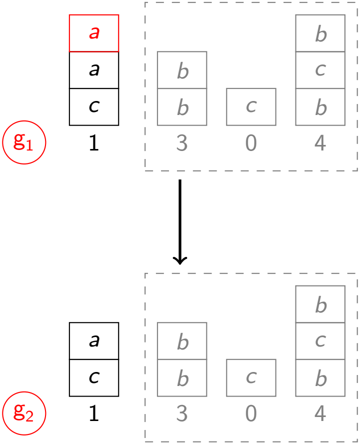
- Initial state g_0,
- an active thread with
  - only the initial symbol γ_0 on the stack
  - and context switch number 0,
- and an empty bag.



Pop, push, and spawn behavior is defined by transition rules.

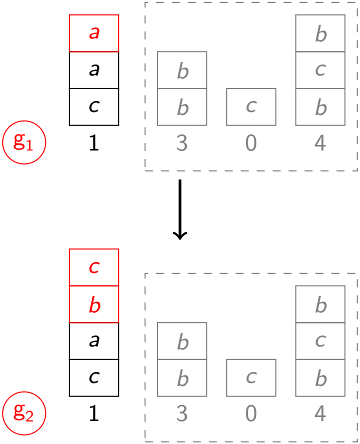
Pop Behavior

Consider transition rule $g_1|a \hookrightarrow g_2|\varepsilon$:



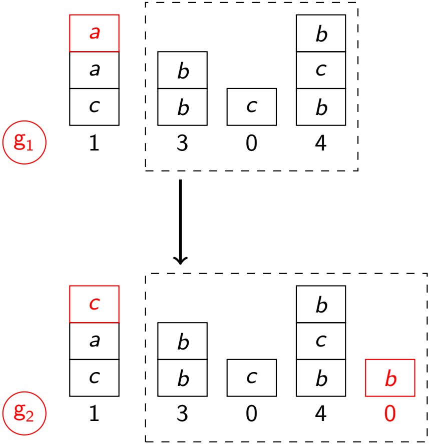
Push Behavior

Consider transition rule $g_1|a \hookrightarrow g_2|cb$:



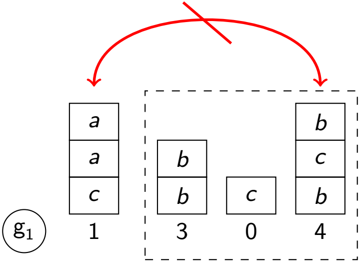
Spawn Behavior

Consider transition rule $g_1|a \hookrightarrow g_2|c \triangleright b$:



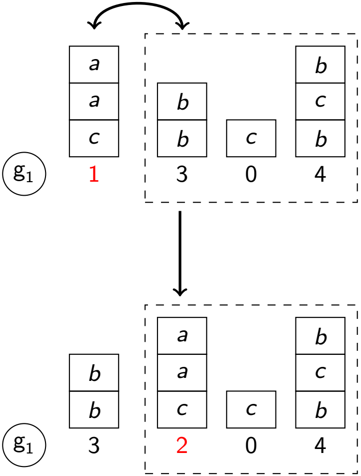
Context Switch Behavior

Consider the context switch bound $K = 3$:



Context Switch Behavior

Consider the context switch bound $K = 3$:



DCPS State Reachability

K-bounded state reachability problem for DCPS (SRP[K])

Input A DCPS $\mathcal{A}$ and a global state g

Question Is g K-bounded reachable in $\mathcal{A}$?

Main result

SRP[K] is 2EXPSPACE-hard for every $K \geq 1$.

Together with Atig, Bouajjani, and Qadeer (2009) this gives us:

Theorem

SRP[K] is 2EXPSPACE-complete for every $K \geq 1$.

2EXPSPACE Lower Bound

Proof Outline

The proof consists of a 3 step reduction:

Termination of triple-exponentially bounded counter programs

$\downarrow$

Termination of recursive net programs (RNP)*

$\downarrow$

Coverability for transducer-defined Petri nets (TDPN)*

$\downarrow$

SRP[1] for DCPS

*new model

Proof Outline

The proof consists of a 3 step reduction:

Termination of triple-exponentially bounded counter programs

$\downarrow$ Adapted Lipton construction

Termination of recursive net programs (RNP)*

$\downarrow$ Adapted Lipton construction

Coverability for transducer-defined Petri nets (TDPN)*

$\downarrow$ New techniques

SRP[1] for DCPS

*new model

Proof Outline

The proof consists of a 3 step reduction:

Termination of triple-exponentially bounded counter programs

↓ Adapted Lipton construction

Termination of recursive net programs (RNP)*

↓ Adapted Lipton construction

Coverability for transducer-defined Petri nets (TDPN)*

↓ New techniques

SRP[1] for DCPS

*new model

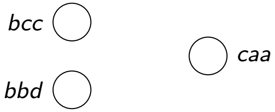
Transducer-Defined Petri Nets (TDPN)

Succinct representation of Petri nets:

Transducer-Defined Petri Nets (TDPN)

Succinct representation of Petri nets:

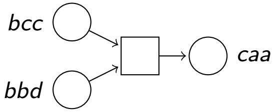
- Each Petri net place is assigned an address (word over an alphabet).



Transducer-Defined Petri Nets (TDPN)

Succinct representation of Petri nets:

- Each Petri net place is assigned an address (word over an alphabet).
- Petri net transitions correspond to vectors of such addresses.



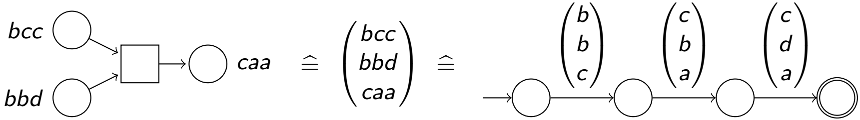
$$\widehat{=} \begin{pmatrix} bcc \\ bbd \\ caa \end{pmatrix}$$

Petri net transition

Transducer-Defined Petri Nets (TDPN)

Succinct representation of Petri nets:

- Each Petri net place is assigned an address (word over an alphabet).
- Petri net transitions correspond to vectors of such addresses.
- These vectors are accepted by transducers.



Petri net transition

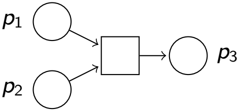
transducer run

Transducer-Defined Petri Nets (TDPN)

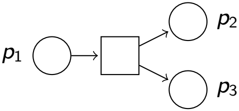
- Issue: These vectors do not distinguish input and output.
- Solution: Three transducers for three types of transitions:

Transducer-Defined Petri Nets (TDPN)

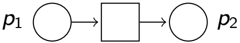
- Issue: These vectors do not distinguish input and output.
- Solution: Three transducers for three types of transitions:



$$\begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \in L(\mathcal{T}_{join})$$



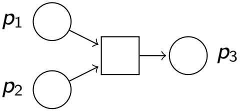
$$\begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \in L(\mathcal{T}_{fork})$$

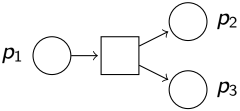


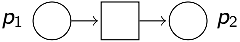
$$\begin{pmatrix} p_1 \\ p_2 \end{pmatrix} \in L(\mathcal{T}_{move})$$

Transducer-Defined Petri Nets (TDPN)

- Issue: These vectors do not distinguish input and output.
- Solution: Three transducers for three types of transitions:







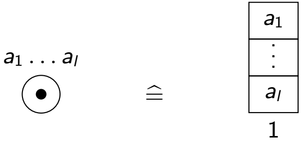
$\begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \in L(\mathcal{T}_{join})$ $\quad \begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \in L(\mathcal{T}_{fork})$ $\quad \begin{pmatrix} p_1 \\ p_2 \end{pmatrix} \in L(\mathcal{T}_{move})$

A TDPN consists of these three transducers plus two addresses:

- w_{init} for the initial marking:  w_{init} ,
- and w_{final} for the marking to cover:  w_{final}

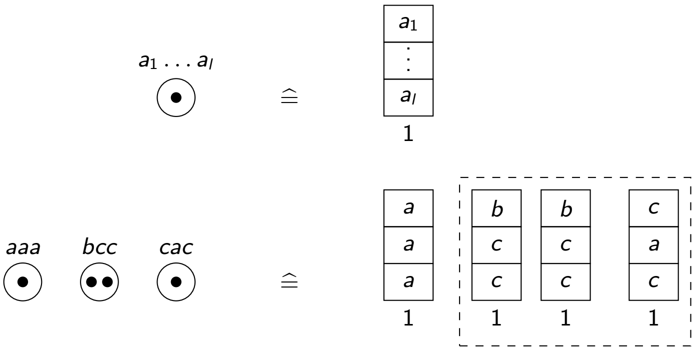
Representing TDPN Tokens via DCPS

A token on place w corresponds to a thread with w on its stack:

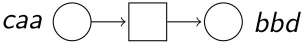


Representing TDPN Tokens via DCPS

A token on place w corresponds to a thread with w on its stack:

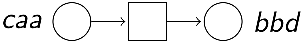


Simulating TDPN Transitions via DCPS: Read Stage

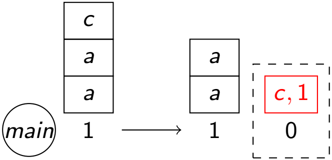


Remove the input token, save address symbols in the bag:

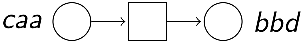
Simulating TDPN Transitions via DCPS: Read Stage



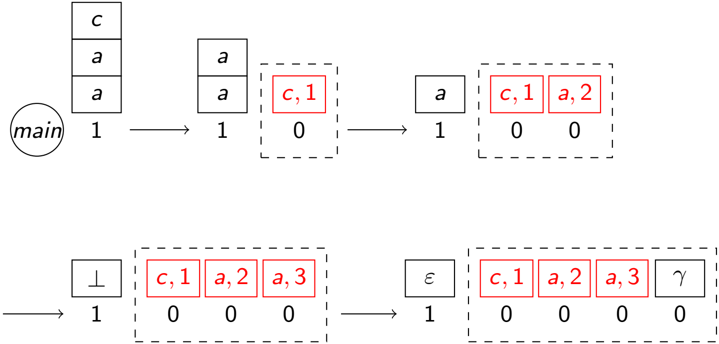
Remove the input token, save address symbols in the bag:



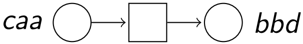
Simulating TDPN Transitions via DCPS: Read Stage



Remove the input token, save address symbols in the bag:

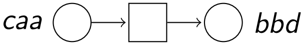


Simulating TDPN Transitions via DCPS: Guess Stage

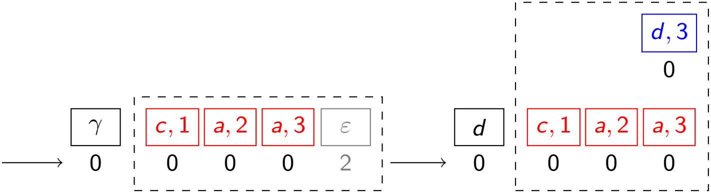


Guess the output token, save address symbols in the bag:

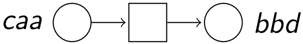
Simulating TDPN Transitions via DCPS: Guess Stage



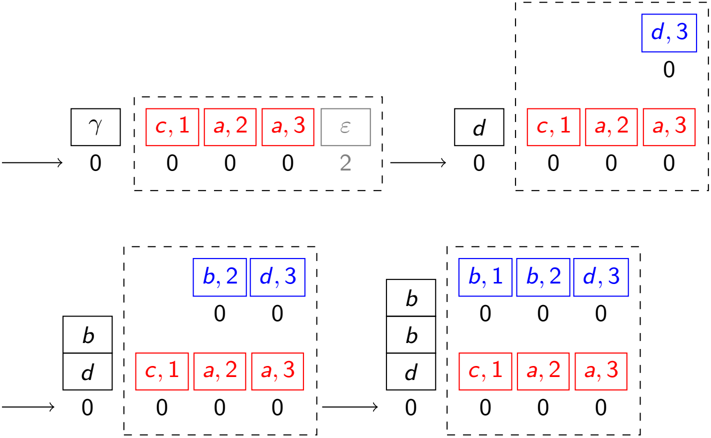
Guess the output token, save address symbols in the bag:



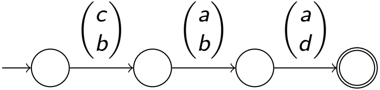
Simulating TDPN Transitions via DCPS: Guess Stage



Guess the output token, save address symbols in the bag:



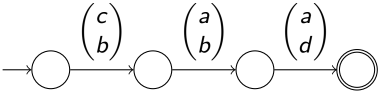
Simulating TDPN Transitions via DCPS: Verify Stage



in $\mathcal{T}_{move}$

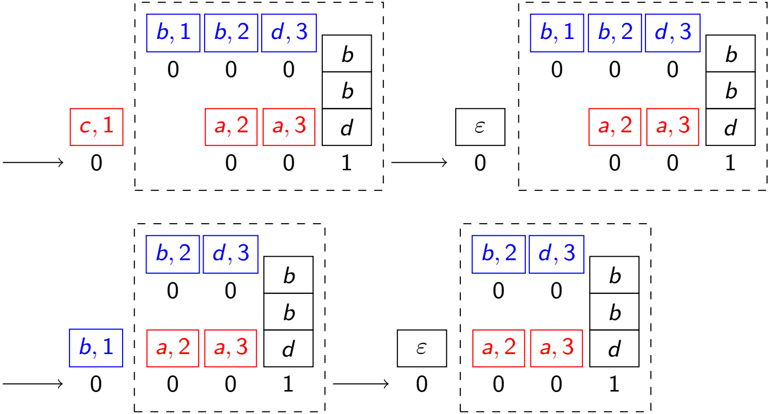
Verify correspondence to such a transducer run:

Simulating TDPN Transitions via DCPS: Verify Stage

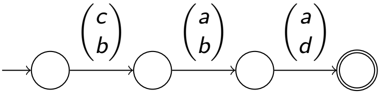


in $\mathcal{T}_{move}$

Verify correspondence to such a transducer run:

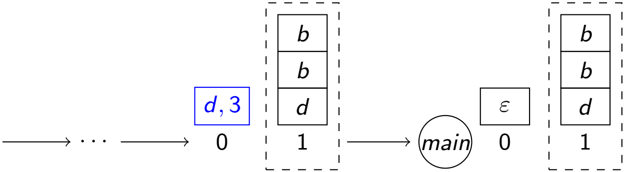


Simulating TDPN Transitions via DCPS: Verify Stage



in $\mathcal{T}_{move}$

From here we continue until:



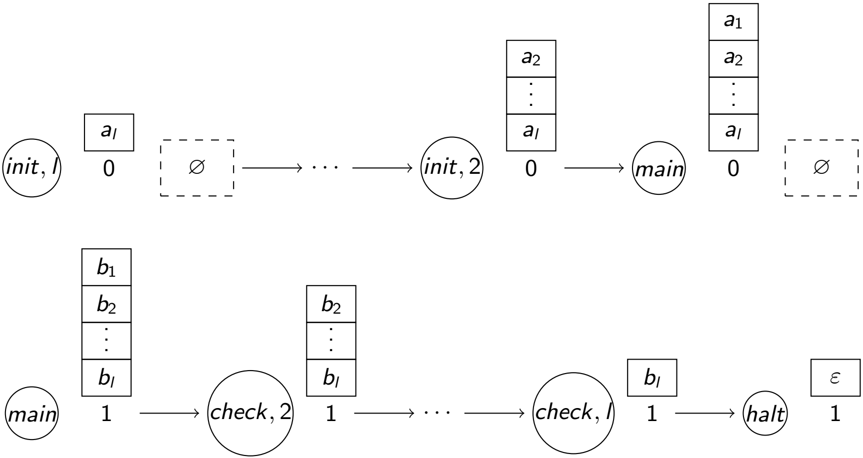
Simulating TDPN Transitions via DCPS

Transitions accepted by the other transducers require slight changes:

- $\mathcal{T}_{join}$ requires another thread's stack to be emptied,
- $\mathcal{T}_{fork}$ requires another new thread's stack to be guessed,
- and both of them need 3 sets of symbol threads in the bag.

Initial and Final Markings

Let $w_{init} = a_1 \dots a_l$ and $w_{final} = b_1 \dots b_l$.



Proof Outline

Termination of triple-exponentially bounded counter programs

Termination of recursive net programs (RNP)

Coverability for transducer-defined Petri nets (TDPN)

$\downarrow \checkmark$

SRP[1] for DCPS

Proof Outline

Termination of triple-exponentially bounded counter programs

Termination of recursive net programs (RNP)

Coverability for transducer-defined Petri nets (TDPN)

✓

SRP[1] for DCPS

Proof Outline

Termination of triple-exponentially bounded counter programs

↓

Termination of recursive net programs (RNP)

Coverability for transducer-defined Petri nets (TDPN)

↓ ✓

SRP[1] for DCPS

Instead of RNP, one could reduce from chain systems to TDPN.

- Introduced by Demri, Figueira and Praveen, 2013.

Original Lipton Construction

Due to Lipton, 1976.

- We considered the explanation by Esparza, 1998.

Original Lipton Construction

Due to Lipton, 1976.

- We considered the explanation by Esparza, 1998.

Shows EXPSPACE-hardness by reducing from bounded counter programs:

- Counters bounded by 2^{2^n} with n given in unary.
- Can the **halt** command be reached?

Original Lipton Construction

Due to Lipton, 1976.

- We considered the explanation by Esparza, 1998.

Shows EXPSPACE-hardness by reducing from bounded counter programs:

- Counters bounded by 2^{2^n} with n given in unary.
- Can the **halt** command be reached?

Reduce to reachability of the **halt** command in net programs:

- Counters cannot be tested for zero.
- To remedy this, keep a complement counter $\bar{x}$ for each counter x.
- Uphold invariant $x + \bar{x} = 2^{2^n}$.
- Test x for zero by checking that $\bar{x}$ is at maximum value.

Original Lipton Construction

We need to perform 2^{2^n} decrements on $\bar{x}$:

- Make use of the equality $2^{2^n} = 2^{2^{n-1}} \cdot 2^{2^{n-1}}$.
- Use two nested loops, each running $2^{2^{n-1}}$ times.
- Decrement a helper variable for each loop, test for zero at the end.
- Gives rise to nested zero tests, from level n down to 0.
- At level 0 decrement $2^{2^0} = 2$ times.

Original Lipton Construction

The program then consists of:

- $n + 1$ decrement subroutines for zero tests.
- $n + 1$ increment subroutines for initializing complement counters.
- $n + 1$ copies of each helper variable.

$$\begin{array}{ccc} \mathtt{dec}_n & \mathtt{inc}_n & y_n,\ z_n \\ \vdots & \vdots & \vdots \\ \mathtt{dec}_0 & \mathtt{inc}_0 & y_0,\ z_0 \end{array}$$

Adapted Lipton Construction

Get rid of duplicate code using recursion:

- Only a single recursive definition per subroutine.
- Implicitly use different variable copies depending on recursion depth.

`dec` `inc` y, z

Adapted Lipton Construction

Get rid of duplicate code using recursion:

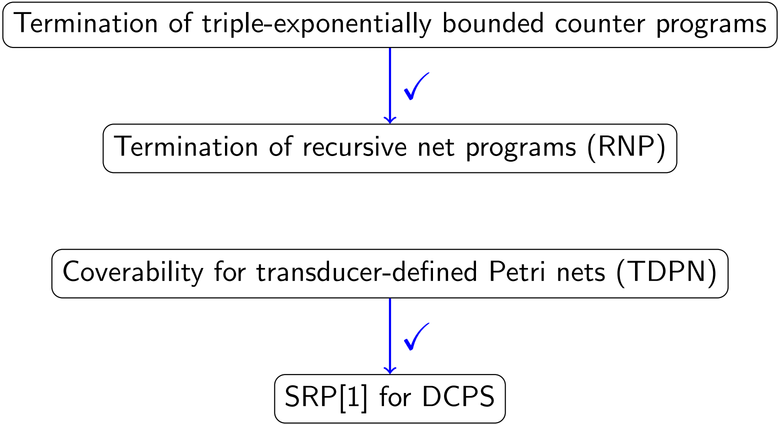
- Only a single recursive definition per subroutine.
- Implicitly use different variable copies depending on recursion depth.

`dec` `inc` y, z

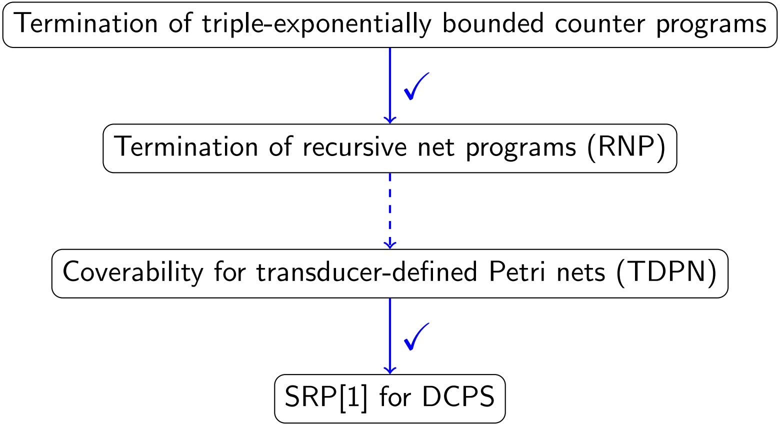
Recursive net programs (RNP) define a bound on their recursion depth.

- For EXPSPACE-hardness, use bound n.
- The bound 2^n would allow for triply exponential counter values.
- We can encode 2^n in poly(input) space, since n was given in unary.
- This then lifts the construction to 2EXPSPACE-hardness.

Proof Outline



Proof Outline



Simulating RNP via TDPN

For each possible recursion depth d from 0 to 2^n:

- one place per counter x,
- and one place per line of code (plus some auxiliary places).

Simulating RNP via TDPN

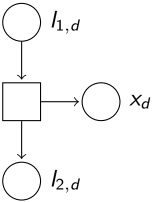
For each possible recursion depth d from 0 to 2^n:

- one place per counter x,
- and one place per line of code (plus some auxiliary places).

Use transitions to simulate the commands:

l_1 : **inc** x;

l_2 : . . .



Simulating RNP via TDPN

For each possible recursion depth d from 0 to 2^n:

- one place per counter x,
- and one place per line of code (plus some auxiliary places).

Use transitions to simulate the commands:

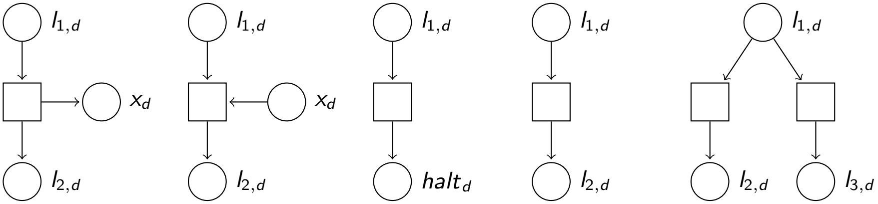
l_1 : **inc** x;
l_2 : . . .

l_1 : **dec** x;
l_2 : . . .

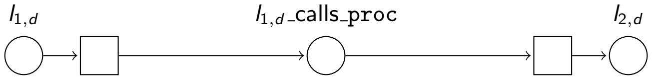
l_1 : **halt**;

l_1 : **goto** l_2;

l_1 : **goto** l_2 **or goto** l_3;



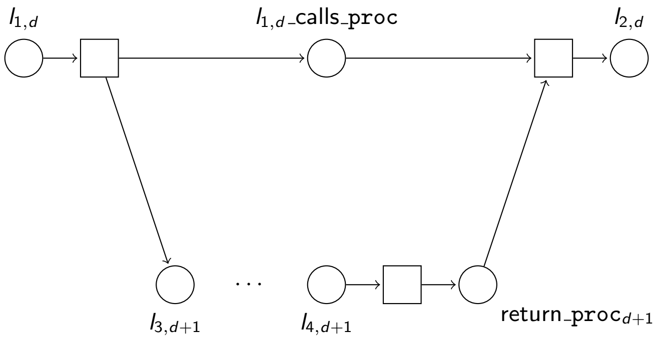
Simulating RNP via TDPN



l_1 : **call** `proc`;

l_2 : ...

Simulating RNP via TDPN



l_1 : **call** `proc`;

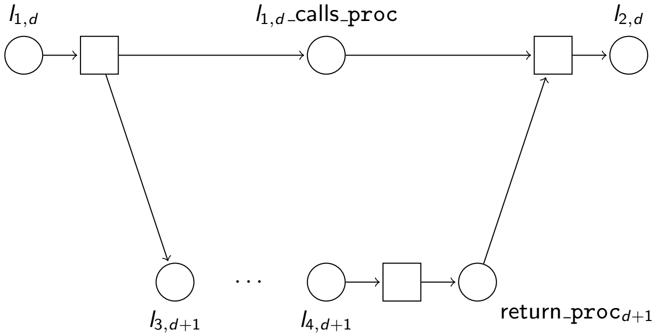
l_2 : ...

`proc`: l_3 : ...

⋮

l_4 : **return**;

Simulating RNP via TDPN



l_1 : **call** `proc`;
l_2 : ...

`proc`: l_3 : ...
⋮
l_4 : **return**;

- Define w_{init} as the first line in the program at depth 0.
- Define w_{final} as the auxiliary halting place at depth 0.

Succinct Representation via Transducers

Use binary addresses $w = u.v$ for places:

- u: Role, i.e. which line, counter, or auxiliary place it is.
- v: Binary representation of recursion depth d.

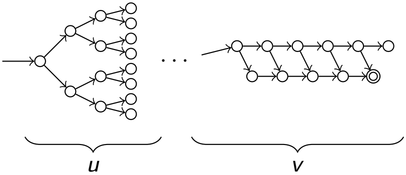
Succinct Representation via Transducers

Use binary addresses $w = u.v$ for places:

- u: Role, i.e. which line, counter, or auxiliary place it is.
- v: Binary representation of recursion depth d.

Construct transducers with polynomially many states:

- There are only polynomially many possibilities for triples of u.
- Comparing binary encodings of d takes polynomial space.



Proof Outline

Termination of triple-exponentially bounded counter programs

$\downarrow$ ✓

Termination of recursive net programs (RNP)

$\downarrow$ ✓

Coverability for transducer-defined Petri nets (TDPN)

$\downarrow$ ✓

SRP[1] for DCPS

Conclusion

Conclusion

Together with previously known results we have shown:

Theorem

SRP[K] is EXPSPACE-complete for $K = 0$,
and 2EXPSPACE-complete for every $K \geq 1$.

There are no remaining complexity gaps!

Conclusion

Together with previously known results we have shown:

Theorem

SRP[K] is EXPSPACE-complete for $K = 0$,
and 2EXPSPACE-complete for every $K \geq 1$.

There are no remaining complexity gaps!

We introduced two new natural models, RNP and TDPN.

- Their 2EXPSPACE-complete problems are of independent interest.

Conclusion

Together with previously known results we have shown:

Theorem

SRP[K] is EXPSPACE-complete for $K = 0$,
and 2EXPSPACE-complete for every $K \geq 1$.

There are no remaining complexity gaps!

We introduced two new natural models, RNP and TDPN.

- Their 2EXPSPACE-complete problems are of independent interest.

Our hardness result can even be applied to an adjacent model:

- Replicated finite-state programs (Kaiser, Kroening, and Wahl, 2010).

Ongoing Work

Study problems for DCPS related to liveness verification:

- Non-termination.
- Fair non-termination (distinguish threads with different make-up).
- Non-starvation (distinguish all threads).

Thank you for your attention!

Sources I

Mohamed Faouzi Atig, Ahmed Bouajjani, and Shaz Qadeer.
Context-bounded analysis for concurrent programs with dynamic creation of threads.
In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 107–123, 2009.

Stéphane Demri, Diego Figueira, and M. Praveen.
Reasoning about data repetitions with counter systems.
In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 33–42, 2013.

Sources II

Javier Esparza.
Decidability and complexity of Petri net problems – an introduction.
In G. Rozenberg and W. Reisig, editors, *Lectures on Petri Nets I: Basic Models. Advances in Petri Nets*, number 1491 in Lecture Notes in Computer Science, pages 374–428, 1998.

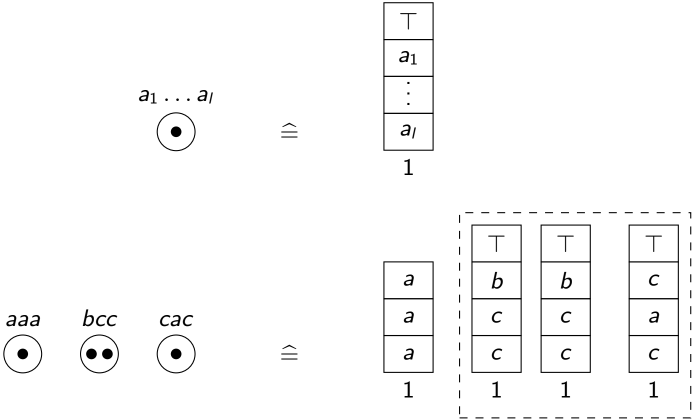
Pierre Ganty and Rupak Majumdar.
Algorithmic verification of asynchronous programs.
ACM Transactions on Programming Languages and Systems (TOPLAS), 34(1):6, 2012.

Alexander Kaiser, Daniel Kroening, and Thomas Wahl.
Dynamic cutoff detection in parameterized concurrent programs.
In *22nd International Conference on Computer Aided Verification, CAV 2010, Edinburgh, UK, July 15-19, 2010, Proceedings*, pages 645–659. Springer, 2010.

Sources III

Richard Lipton.
The reachability problem is exponential-space hard.
Yale University, Department of Computer Science, Report, 62, 1976.

Locking Inactive Threads



Lifting to 2EXPSPACE

We used $2^{2^d} = 2^{2^{d-1} \cdot 2} = \left(2^{2^{d-1}}\right)^2 = 2^{2^{d-1}} \cdot 2^{2^{d-1}}$.

- This means from one level to the next the bound gets squared.

$$(\cdots(2\overbrace{{}^{2})^{2}\cdots)^{2}}^{n\text{-times}} = 2^{2^n}$$

$$(\cdots(2\overbrace{{}^{2})^{2}\cdots)^{2}}^{2^n\text{-times}} = 2^{2^{2^n}}$$

Details of Known Results

Ganty and Majumdar (2012) consider threads running to completion.

- We can ensure that threads empty their stack in our model.
- This allows us to use their EXPSPACE-completeness result for $K = 0$.

Atig, Bouajjani, and Qadeer (2009) consider a slightly different DCPS:

- Each thread spawns with its parents cs-number plus 1.
- We can simulate our model in theirs using 2 more contex switches.
- Reduces our SRP[K] to their SRP[$K + 2$].
- This allows us to use their 2EXPSPACE-membership result.

Succinct Representation via Transducers

Use binary addresses $w = u.v$ for places:

- u: Role, i.e. which line, counter, or auxiliary place it is.
- v: Binary representation of recursion depth d.

Let the size of the RNP be h, the number of lines of code.

- Each counter appears in at least one line.
- Each line only needs at most one auxiliary place.
- Thus, the number of possibilities for u is linear in h.

Make the transducers distinguish each possible triple (pair) of prefixes u:

- Considering triples adds an exponent of 3, still poly in h.

Succinct Representation via Transducers

The recursion depth d changes by at most 1 at a time.

- Transducers have to check for equality or off-by-one on postfixes v.
- These checks require space linear in the number of bits.
- Since the maximum for d is 2^n, v has $\log(2^n) = n$ bits.

The triple (pair) of prefixes u tells us how the depths are related.

- Connect the paths for u with the appropriate checks at the end.