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);
    } elseif * {
        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, accessible 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, accessible 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 $\gamma_0$ 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 $\gamma_0$ 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 $g_1|a \leftrightarrow g_2|\varepsilon$:
Push Behavior

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

$$
\begin{array}{|c|c|c|}
\hline
a & b & b \\
\hline
a & c & c \\
\hline
\end{array}
\begin{array}{|c|c|c|}
\hline
b & c & b \\
\hline
b & 0 & 4 \\
\hline
\end{array}
$$
Spawn Behavior

Consider transition rule $g_1 \cd a \leftrightarrow g_2 \cd c \triangleright b$:
Context Switch Behavior

Consider the context switch bound $K = 3$:
Context Switch Behavior

Consider the context switch bound $K = 3$: 

\[
\begin{array}{c}
  a \\
  a \\
  c \\
\end{array}
\quad 1

\begin{array}{c}
  b \\
  b \\
  c \quad 0 \\
\end{array}
\quad 3

\begin{array}{c}
  b \\
  c \quad 0 \\
\end{array}
\quad 4

\begin{array}{c}
  b \\
  b \\
  c \\
\end{array}
\quad 2

\begin{array}{c}
  a \\
  a \\
  c \\
\end{array}
\quad 0

\begin{array}{c}
  b \\
  c \quad 0 \\
\end{array}
\quad 4

\begin{array}{c}
  b \\
  b \\
\end{array}
\quad 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:

1. Termination of triple-exponentially bounded counter programs
2. Termination of recursive net programs (RNP)*
3. Coverability for transducer-defined Petri nets (TDPN)*
4. SRP[1] for DCPS

*new model
Proof Outline

The proof consists of a 3 step reduction:

1. **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
Proof Outline

The proof consists of a 3 step reduction:

1. Termination of triple-exponentially bounded counter programs
2. Termination of recursive net programs (RNP)*
3. Coverability for transducer-defined Petri nets (TDPN)*
   - Adapted Lipton construction
   - 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).

\[
\begin{array}{c}
\text{bcc} \\
\text{bbd}
\end{array}
\]

\[
\begin{array}{c}
\bigcirc \\
\bigcirc \text{caa}
\end{array}
\]
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.

\[
\begin{pmatrix}
  bcc \\
  bbd
\end{pmatrix} \rightarrow
\begin{pmatrix}
  ccc \\
  bbd
\end{pmatrix} 
\Rightarrow \begin{pmatrix}
  bcc \\
  bbd \\
  cca
\end{pmatrix}
\]

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.

\[
\begin{pmatrix}
\text{bcc} \\
\text{bbd}
\end{pmatrix} \rightarrow \begin{pmatrix}
\text{caa} \\
\end{pmatrix} \cong \begin{pmatrix}
\text{bcc} \\
\text{bbd} \\
\text{caa}
\end{pmatrix} \cong \begin{pmatrix}
\text{b} \\
\text{b} \\
\text{c}
\end{pmatrix} \rightarrow \begin{pmatrix}
\text{c} \\
\text{b} \\
\text{a}
\end{pmatrix} \rightarrow \begin{pmatrix}
\text{c} \\
\text{d} \\
\text{a}
\end{pmatrix}
\]
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:

\[
\begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \in L(T_{\text{join}}) \\
\begin{pmatrix} p_1 \\ p_2 \\ p_3 \end{pmatrix} \in L(T_{\text{fork}}) \\
\begin{pmatrix} p_1 \\ p_2 \end{pmatrix} \in L(T_{\text{move}})
\]
Transducer-Defined Petri Nets (TDPN)

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

  
  ![Diagram of TDPN transitions]

  
  \[
  \begin{pmatrix}
  p_1 \\
  p_2 \\
  p_3 
  \end{pmatrix} \in L(T_{\text{join}}) \\
  \begin{pmatrix}
  p_1 \\
  p_2 \\
  p_3 
  \end{pmatrix} \in L(T_{\text{fork}}) \\
  \begin{pmatrix}
  p_1 \\
  p_2 
  \end{pmatrix} \in L(T_{\text{move}})
  \]

A TDPN consists of these three transducers plus two addresses:

- **w_{init}** for the initial marking: \( \bigcirc \) \( w_{init} \),
- and **w_{final}** for the marking to cover: \( \bigcirc \) \( w_{final} \).
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:

$$a_1 \ldots a_l$$

$
\begin{array}{c}
\bullet \\
\end{array}$$

$\equiv$

$$a_1$$

$
\begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
\bullet \\
\end{array}$$

$\equiv$

$$\begin{array}{c}a \\
a \\
a \\
1 \end{array}$$

$\begin{array}{c}b \\
c \\
c \\
1 \end{array}$

$\begin{array}{c}b \\
c \\
c \\
1 \end{array}$

$\begin{array}{c}c \\
a \\
c \\
1 \end{array}$
Simulating TDPN Transitions via DCPS: Read Stage

${caa \xrightarrow{} \square \xrightarrow{} \bigcirc bbd}$

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:

\[ \text{caa} \rightarrow \square \rightarrow \text{bbd} \]
Simulating TDPN Transitions via DCPS: Read Stage

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

\[
\begin{align*}
\text{caa} & \rightarrow \square \rightarrow \text{bbd} \\
\text{main} & \rightarrow 1 \\
& \rightarrow 1 \\
& \rightarrow 0 \\
& \rightarrow 1 \\
& \rightarrow 0 \\
& \rightarrow \text{c,1} \\
& \rightarrow \text{a,2} \\
& \rightarrow \text{a,3} \\
& \rightarrow \varepsilon \\
& \rightarrow \gamma
\end{align*}
\]
Simulating TDPN Transitions via DCPS: Guess Stage

\[ caa \xrightarrow{} \square \xrightarrow{} bbd \]

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:

\[ \text{caa} \xrightarrow{} \text{□} \xrightarrow{} \text{□} \text{ bbd} \]
Simulating TDPN Transitions via DCPS: Guess Stage

\[

caa \rightarrow \square \rightarrow bbd
\]

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

\[
\begin{array}{c}
\gamma \\
0
\end{array} \quad \begin{array}{cccc}
c, 1 & a, 2 & a, 3 & \varepsilon \\
0 & 0 & 0 & 2
\end{array} \quad \begin{array}{c}
d \\
0
\end{array} \quad \begin{array}{cccc}
c, 1 & a, 2 & a, 3 & d, 3 \\
0 & 0 & 0 & 0
\end{array}
\]

\[
\begin{array}{c}
b \\
0
\end{array} \quad \begin{array}{cccc}
c, 1 & a, 2 & a, 3 & d, 3 \\
0 & 0 & 0 & 0
\end{array} \quad \begin{array}{c}
b \\
0
\end{array} \quad \begin{array}{cccc}
c, 1 & a, 2 & a, 3 & d, 3 \\
0 & 0 & 0 & 0
\end{array}
\]
Simulating TDPN Transitions via DCPS: Verify Stage

Verify correspondence to such a transducer run:

\[
\begin{align*}
(c, b) & \rightarrow (a, b) \rightarrow (a, d) \rightarrow \epsilon \\
\text{in } T_{move}
\end{align*}
\]
Simulating TDPN Transitions via DCPS: Verify Stage

Verify correspondence to such a transducer run:
Simulating TDPN Transitions via DCPS: Verify Stage

From here we continue until:

\[ \begin{array}{c}
\begin{array}{c}
\text{move}
\end{array}
\end{array} \]
Simulating TDPN Transitions via DCPS

Transitions accepted by the other transducers require slight changes:

- $T_{\text{join}}$ requires another thread’s stack to be emptied,
- $T_{\text{fork}}$ 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 $w_{\text{init}} = a_1 \ldots a_l$ and $w_{\text{final}} = b_1 \ldots b_l$. 
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.
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.
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 \texttt{halt} command be reached?

Reduce to reachability of the \texttt{halt} command in net programs:

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

\[
\begin{align*}
\text{dec}_n & \quad \text{inc}_n & \quad y_n, z_n \\
\vdots & & \vdots \\
\text{dec}_0 & \quad \text{inc}_0 & \quad y_0, z_0
\end{align*}
\]
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.

\[
\text{dec} \quad \text{inc} \quad y, z
\]
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.

![Image of 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

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
  - ✓
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:

$l_1 : \text{inc } x;$
$l_2 : \ldots$

$l_1, d$

$\xrightarrow{} x_d$

$l_2, d$
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:

- $l_1 : \text{inc } x$;
- $l_1 : \text{dec } x$;
- $l_1 : \text{halt}$;
- $l_1 : \text{goto } l_2$;
- $l_1 : \text{goto } l_2$ or $\text{goto } l_3$;

- $l_2 : \ldots$
- $l_2 : \ldots$
- $l_2 : \ldots$
- $l_2 : \ldots$
- $l_2 : \ldots$

[Diagram of place and transition labels]
Simulating RNP via TDPN

\[ l_{1,d} \rightarrow l_{1,d-\text{calls}_{\text{proc}}} \rightarrow l_{2,d} \]

\[ l_1 : \text{call proc;} \]
\[ l_2 : \ldots \]
Simulating RNP via TDPN

\[ l_{1,d} \rightarrow l_{1,d-\text{calls_proc}} \rightarrow l_{2,d} \]

\[ l_{3,d+1} \rightarrow \ldots \rightarrow l_{4,d+1} \rightarrow \text{return\_proc}_{d+1} \]

\[ l_1 : \text{call proc;} \]
\[ l_2 : \ldots \]

\[ \text{proc: } l_3 : \ldots \]
\[ \vdots \]
\[ l_4 : \text{return;} \]
Simulating RNP via TDPN

- 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

✓

Termination of recursive net programs (RNP)

✓

Coverability for transducer-defined Petri nets (TDPN)

✓

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 II


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

\[ a_1 \ldots a_l \cong \]

\[ c_a \]

\[ bcc \]

\[ cac \]

\[ \hat{=} \]

\[ \begin{array}{c}
\top \\
a_1 \\
\vdots \\
a_l \\
1 \\
\end{array} \]

\[ \begin{array}{c}
a \\
b \\
c \\
1 \\
1 \\
\end{array} \]
Lifting to $2\text{EXPSPACE}$

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

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

\[
\underbrace{(\ldots (2^2) \ldots )^2}_{\text{n-times}} = 2^{2^n} \\
\underbrace{(\ldots (2^2) \ldots )^2}_{\text{2^n-times}} = 2^{2^{2n}}
\]
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 context 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 \( h \).
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.