On Model-checking Pushdown System Models

PhD defence - July 5, 2018

Adrien Pommellet

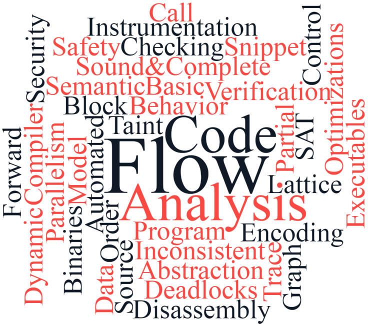
Université Paris-Diderot, IRIF, and LIPN

Thesis directed by **Tayssir Touili**.

Analysing programs

As the complexity of software grows, identifying errors in programs becomes harder and harder.

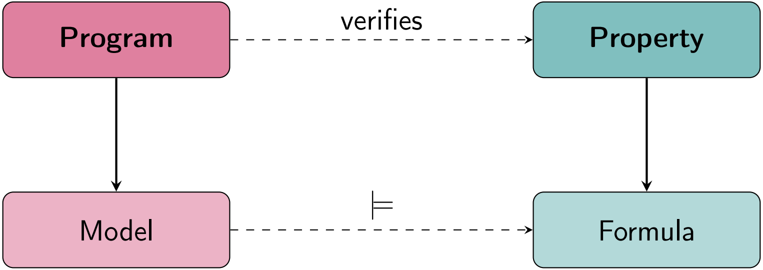
Analysing programs



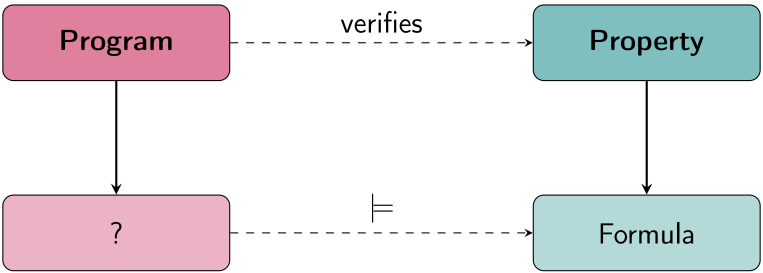
Designing sound and efficient program analysis methods is therefore a matter of the utmost importance.

The model-checking framework

The model-checking framework

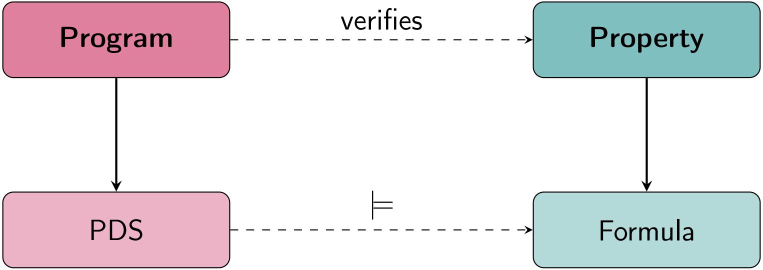


The model-checking framework



Pushdown systems (PDSs) are a natural model for sequential programs [Esparza, Hansel, Rossmanith, and Schwoon, CAV'00] with recursive procedure calls, as they can simulate the stack of a program.

$$p \; \boxed{\gamma_1 \mid \gamma_2 \mid \gamma_3}$$


Program
verifies
Property
PDS
⊨
Formula

Contributions of this thesis

1. We consider the HyperLTL model-checking problem for pushdown systems, we prove that it is unfortunately undecidable, we introduce constraints to regain decidability, then we use these to design under and over-approximation algorithms.

Contributions of this thesis

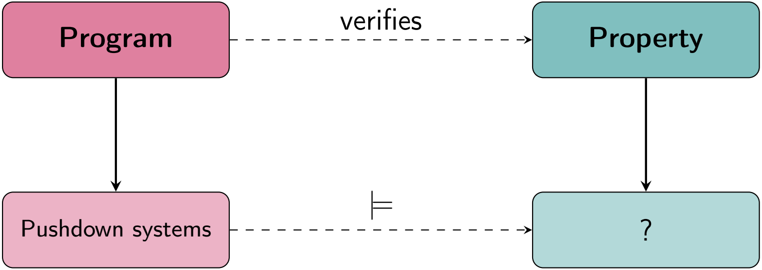
1. We consider the HyperLTL model-checking problem for pushdown systems, we prove that it is unfortunately undecidable, we introduce constraints to regain decidability, then we use these to design under and over-approximation algorithms.
2. We define a new PDS model, called pushdown system with an upper stack (UPDS), that keeps track of the part of the assembly stack that is above the stack pointer, and we propose reachability algorithms for this model.

Contributions of this thesis

1. We consider the HyperLTL model-checking problem for pushdown systems, we prove that it is unfortunately undecidable, we introduce constraints to regain decidability, then we use these to design under and over-approximation algorithms.
2. We define a new PDS model, called pushdown system with an upper stack (UPDS), that keeps track of the part of the assembly stack that is above the stack pointer, and we propose reachability algorithms for this model.
3. We introduce synchronized dynamic pushdown networks (SDPNs) that model concurrent programs as a network of pushdown systems, where each pushdown component can spawn new threads and synchronize by rendez-vous with other threads. We then propose reachability algorithms for this model.

Our first contribution:
HyperLTL model-checking for pushdown systems

The model-checking framework



The logics LTL and CTL may not suffice to express all interesting properties.

Expressing a property

We want that, for every trace π_1, there exists a trace π_2 such that, whenever a property a holds for π_1, property b holds for π_2 at the same step.

$$\pi_1 : \ldots \to a_0 \to y_1 \to a_2 \to z_3 \to \ldots$$

$$\pi_2 : \ldots \to b_0 \to v_1 \to b_2 \to w_3 \to \ldots$$

Expressing a property

We want that, for every trace π_1, there exists a trace π_2 such that, whenever a property a holds for π_1, property b holds for π_2 at the same step.

$$\pi_1 : \ldots \to a_0 \to y_1 \to a_2 \to z_3 \to \ldots$$

$$\pi_2 : \ldots \to b_0 \to v_1 \to b_2 \to w_3 \to \ldots$$

This property cannot be expressed by LTL nor CTL.

Expressing a property

We want that, for every trace π_1, there exists a trace π_2 such that, whenever a property a holds for π_1, property b holds for π_2 at the same step.

$$\pi_1 : \ldots \to a_0 \to y_1 \to a_2 \to z_3 \to \ldots$$

$$\pi_2 : \ldots \to b_0 \to v_1 \to b_2 \to w_3 \to \ldots$$

We would like to express it this way:

$$\psi = \forall \pi_1 \in \mathit{Traces}, \exists \pi_2 \in \mathit{Traces}, \ \mathrm{G}\ (a_{\pi_1} \implies b_{\pi_2})$$

The logic HyperLTL

We want that, for every trace π_1, there exists a trace π_2 such that, whenever a property a holds for π_1, property b holds for π_2 at the same step.

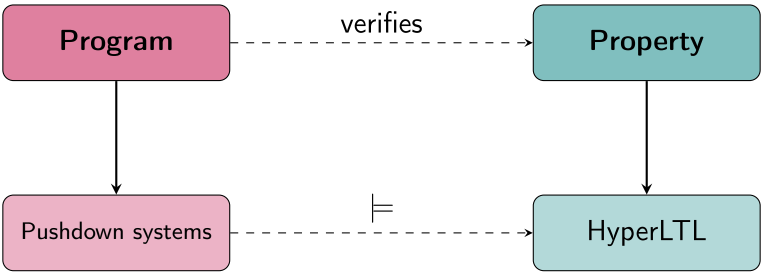
$$\psi = \forall \pi_1 \in \mathit{Traces}, \exists \pi_2 \in \mathit{Traces},\ \mathrm{G}\ (a_{\pi_1} \implies b_{\pi_2})$$

This is actually a HyperLTL formula, where HyperLTL is a logic that extends LTL with the universal and existential quantifications of multiple path variables.

Our first goal

HyperLTL model-checking for finite-state systems has already been solved in [Clarkson et al., POST'14]. But what about pushdown systems?

Our first goal



Defining pushdown systems

Definition

A pushdown system (PDS) is a tuple $\mathcal{P} = (P, \Sigma, \Gamma, \Delta, c_0)$ such that:

- P is a finite set of control states;
- $\Sigma = 2^{AP}$ a finite input alphabet, where AP is a finite set of atomic propositions;
- Γ a finite stack alphabet;
- a finite set Δ of transition rules of the form $(p, \gamma) \xrightarrow{a} (p', w)$;
- $c_0 = \langle p_0, w_0 \rangle$ an initial configuration in $P \times \Gamma^*$.

From $(p, \gamma) \xrightarrow{a} (p', w) \in \Delta$, we infer a transition relation on configurations: $\forall w' \in \Gamma^*, \langle p, \gamma w' \rangle \xrightarrow{a}_{\mathcal{P}} \langle p', ww' \rangle$.

HyperLTL formulas can be used to synchronize traces of pushdown systems.

$$\psi = \forall \pi_1 \in \mathit{Traces}_1, \forall \pi_2 \in \mathit{Traces}_2, (a_{\pi_1} \Leftrightarrow a_{\pi_2})$$

HyperLTL formulas can be used to synchronize traces of pushdown systems. And traces of PDSs are context-free.

$$\psi = \forall \pi_1 \in \overbrace{\mathit{Traces}_1}^{\mathit{CFL}}, \forall \pi_2 \in \overbrace{\mathit{Traces}_2}^{\mathit{CFL}}, (a_{\pi_1} \Leftrightarrow a_{\pi_2})$$

Model-checking HyperLTL for pushdown systems

HyperLTL formulas can be used to synchronize traces of pushdown systems. And traces of PDSs are context-free.

$$\psi = \forall \pi_1 \in \overbrace{Traces_1}^{CFL}, \forall \pi_2 \in \overbrace{Traces_2}^{CFL}, (a_{\pi_1} \Leftrightarrow a_{\pi_2})$$

But the emptiness of the intersection of two context-free languages (CFLs) is well-known to be an undecidable problem.

HyperLTL formulas can be used to synchronize traces of pushdown systems. And traces of PDSs are context-free.

$$\psi = \forall \pi_1 \in \overbrace{Traces_1}^{CFL}, \forall \pi_2 \in \overbrace{Traces_2}^{CFL}, (a_{\pi_1} \Leftrightarrow a_{\pi_2})$$

But the emptiness of the intersection of two context-free languages (CFLs) is well-known to be an undecidable problem. Hence:

Theorem

The model-checking problem of HyperLTL for pushdown systems is undecidable.

Model-checking HyperLTL for visibly pushdown systems

The input-driven sub-class of visibly pushdown systems [Alur et al., STOC'04] is such that we can decide the emptiness of the intersection of two visibly context-free languages.

The input-driven sub-class of visibly pushdown systems [Alur et al., STOC'04] is such that we can decide the emptiness of the intersection of two visibly context-free languages.

However, this constraint is not enough to regain decidability:

Theorem

The model-checking problem of HyperLTL for visibly pushdown systems is undecidable.

With a single context-free variable

Let Reg be a regular language and CFL a context-free language.
Intuitively, we know that we can decide $CFL \cap Reg = \emptyset$.

With a single context-free variable

Let Reg be a regular language and CFL a context-free language.
Intuitively, we know that we can decide $CFL \cap Reg = \emptyset$.

We can prove that:

Theorem

We can decide formulas of the form:
$\psi = \{\forall, \exists\}\pi_1 \in CFL, \{\forall, \exists\}\pi_2 \in Reg_2, \ldots, \{\forall, \exists\}\pi_n \in Reg_n, \varphi.$

Approximating the model-checking problem

If α is a regular over-approximation of the set of traces of a PDS, we therefore can decide:

$$\psi = \{\forall, \exists\}\pi_1 \in \mathit{Traces}, \forall\pi_2 \in \alpha, \ldots, \forall\pi_n \in \alpha, \varphi$$

If ψ holds, then this formula holds as well:

$$\psi' = \{\forall, \exists\}\pi_1 \in \mathit{Traces}, \forall\pi_2 \in \mathit{Traces}, \ldots, \forall\pi_n \in \mathit{Traces}, \varphi$$

In a similar manner, if α is a regular under-approximation of the set of traces of a PDS, we therefore can decide:

$$\psi = \{\forall, \exists\}\pi_1 \in \mathit{Traces}, \exists\pi_2 \in \alpha, \ldots, \exists\pi_n \in \alpha, \varphi$$

If ψ doesn't hold, then this formula does not hold as well:

$$\psi' = \{\forall, \exists\}\pi_1 \in \mathit{Traces}, \exists\pi_2 \in \mathit{Traces}, \ldots, \exists\pi_n \in \mathit{Traces}, \varphi$$

Our first contribution

- We showed that the model-checking problem of HyperLTL for pushdown systems and visibly pushdown systems is undecidable.

Our first contribution

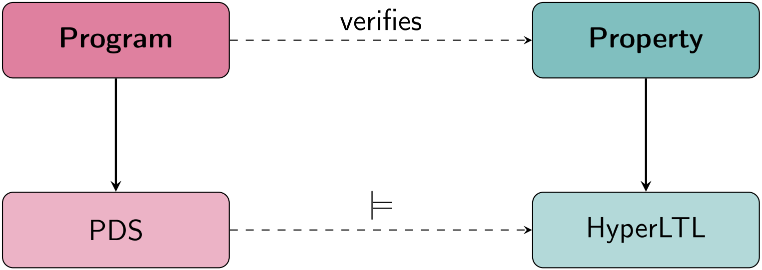
- We showed that the model-checking problem of HyperLTL for pushdown systems and visibly pushdown systems is undecidable.
- We can decide HyperLTL formulas if all variables are regular except the first.

Our first contribution

- We showed that the model-checking problem of HyperLTL for pushdown systems and visibly pushdown systems is undecidable.
- We can decide HyperLTL formulas if all variables are regular except the first.
- We can therefore approximate the answer to the model-checking problem given some constraints on the use of quantifiers in HyperLTL formulas.

Our second contribution:
Reachability analysis of pushdown systems with an upper stack

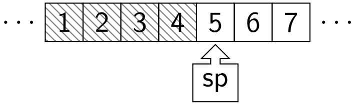
The model-checking framework



Are pushdown systems accurate enough?

Pushdown systems (PDSs) can fail to accurately represent the actual assembly stack.

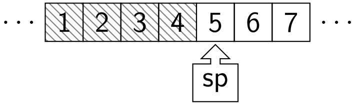
The assembly stack



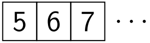
The limits of pushdown systems

Pushdown systems (PDSs) can fail to accurately represent the actual assembly stack.

The assembly stack



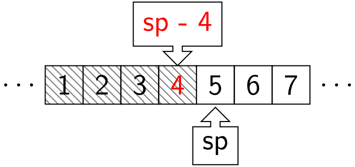
The pushdown model



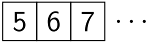
PDSs can't model the part of the assembly stack that stands to the left of the stack pointer.

How can we handle the assembly instruction *mov eax* $[sp - 4]$?

The assembly stack



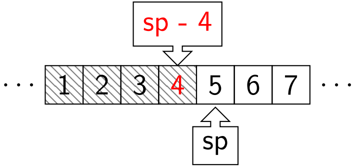
The pushdown model



A new model

How can we handle the assembly instruction *mov eax [sp − 4]*?

The assembly stack



Our new model

Our intuition is to use another stack to model the memory section left of the stack pointer.

Definition

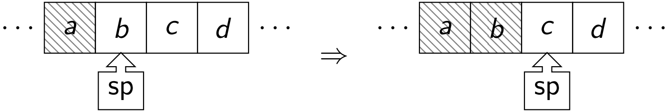
A *pushdown system with an upper stack* (UPDS) is a triplet $\mathcal{P} = (P, \Gamma, \Delta)$ where:

- P is a finite set of control states;
- Γ is a finite stack alphabet;
- a finite set Δ of transition rules of the form $(p, \gamma) \rightarrow (p', w)$, $w \in \Gamma^{\leq 2}$;

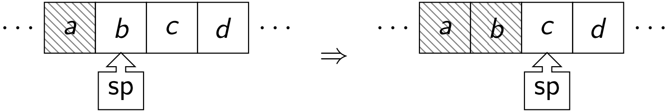
We consider configurations of the form $\langle p, w_u, w_l \rangle$, with a write-only upper stack that accurately models the left of the assembly stack.

Semantics of pop rules

A pop rule in the assembly stack amounts to:



A pop rule in the assembly stack amounts to:

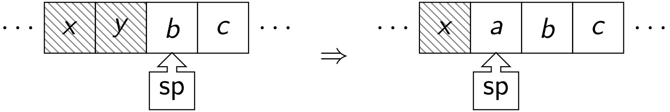


Hence, for a pop rule $\delta = (p, b) \rightarrow (p', \varepsilon)$ in the UPDS:

$$\boxed{a}\ \mathbf{p}\ \boxed{b \mid c \mid d} \overset{\delta}{\Rightarrow} \boxed{a \mid b}\ \mathbf{p'}\ \boxed{c \mid d}$$

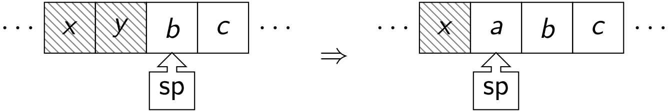
Semantics of push rules

A push rule in the assembly stack amounts to:



Semantics of push rules

A push rule in the assembly stack amounts to:



For a push rule $\delta = (p, b) \to (p', ab)$ in the UPDS:

$$\boxed{x}\boxed{y}\ \mathbf{p}\ \boxed{b}\boxed{c} \stackrel{\delta}{\Rightarrow} \boxed{x}\ \mathbf{p'}\ \boxed{a}\boxed{b}\boxed{c}$$

Are the sets of predecessors pre^* and successors $post^*$ of a regular set of configurations of a UPDS regular and effectively computable, in a manner similar to PDSs?

Reachability properties of UPDSs

Theorem

There exist a UPDS $\mathcal{P}$ and a regular set of configurations $\mathcal{C}$ for which $post^* (\mathcal{C})$ is not regular.

Theorem

There exist a UPDS $\mathcal{P}$ and a regular set of configurations $\mathcal{C}$ for which $pre^* (\mathcal{C})$ is not regular.

Theorem

Given a UPDS $\mathcal{P}$ and a regular set of configurations $\mathcal{C}$, $post^* (\mathcal{C})$ is context-sensitive, and its membership problem is therefore decidable.

The set of runs of a UPDS, being similar to a PDS's, is context-free. But what if this set is regular?

Theorem

For a UPDS $\mathcal{P} = (P, \Gamma, \Delta)$, a regular set of configurations $\mathcal{C}$, and a regular set of runs R of $\mathcal{P}$ from $\mathcal{C}$, the set of upper stack configurations reachable using runs in R is regular and effectively computable.

Computing a regular over-approximation of *post**

1. Compute a regular over-approximation R of the set of runs of the PDS $\mathcal{P}$ from $\mathcal{C}$;

Computing a regular over-approximation of $post^*$

1. Compute a regular over-approximation R of the set of runs of the PDS $\mathcal{P}$ from $\mathcal{C}$;
2. compute the set U of upper stack configurations reachable using R of $\mathcal{P}$;

1. Compute a regular over-approximation R of the set of runs of the PDS $\mathcal{P}$ from $\mathcal{C}$;
2. compute the set U of upper stack configurations reachable using R of $\mathcal{P}$;
3. compute the exact set L of reachable lower stack configurations, using a standard reachability algorithm for PDSs [Esparza, Schwoon et al., CAV'00][Caucal,'92];

Computing a regular over-approximation of *post**

1. Compute a regular over-approximation R of the set of runs of the PDS $\mathcal{P}$ from $\mathcal{C}$;
2. compute the set U of upper stack configurations reachable using R of $\mathcal{P}$;
3. compute the exact set L of reachable lower stack configurations, using a standard reachability algorithm for PDSs [Esparza, Schwoon et al., CAV'00][Caucal,'92];
4. consider the product $U \times L$ of the upper and lower stack sets to create an over-approximation of $post^*(\mathcal{C})$.

- A UPDS can be simulated by a multi-stack pushdown system (MPDS) with two stacks.

Computing a regular under-approximation of *pre**

- A UPDS can be simulated by a multi-stack pushdown system (MPDS) with two stacks.
- But the set of predecessors of a MPDS can be under-approximated, using a phase-bounding constraint [Seth, CAV'10].

Computing a regular under-approximation of pre^*

- A UPDS can be simulated by a multi-stack pushdown system (MPDS) with two stacks.
- But the set of predecessors of a MPDS can be under-approximated, using a phase-bounding constraint [Seth, CAV'10].
- Hence, we can under-approximate the set of predecessors pre^* of a UPDS.

Applications

Stack overflow detection

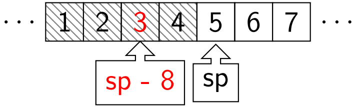
Application 1

We want to prevent the stack from growing beyond a bound $m+1$.
We put a symbol $\top$ on top of an upper stack of bounded height m filled with $\#$ padding symbols.

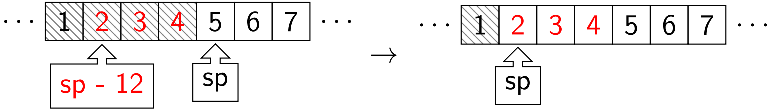
$$\boxed{\top}\underbrace{\boxed{\#}\cdots\boxed{\#}}_{m \text{ times}}\;\boxed{a}$$

If the symbol $\top$ is overwritten, we deduce that a stack overflow malfunction happens.

A register is assigned a value located in the upper stack: the instruction *mov eax* $[sp - 8]$ copies in the register *eax* the second symbol above the stack pointer *sp*.



If we apply the instruction *sub sp* 12, we change the stack pointer *sp*, leading to a new stack configuration:



Our second contribution

- We defined a new automaton model, called UPDS, that models the stack of a program more accurately than a PDS.

Our second contribution

- We defined a new automaton model, called UPDS, that models the stack of a program more accurately than a PDS.
- We show that the backward and forward reachability sets of UPDSs are not regular, but that the latter is context-sensitive.

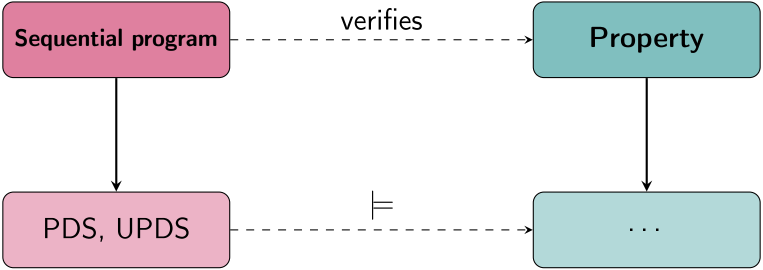
- We defined a new automaton model, called UPDS, that models the stack of a program more accurately than a PDS.
- We show that the backward and forward reachability sets of UPDSs are not regular, but that the latter is context-sensitive.
- We can either under-approximate or over-approximate these sets.

Our second contribution

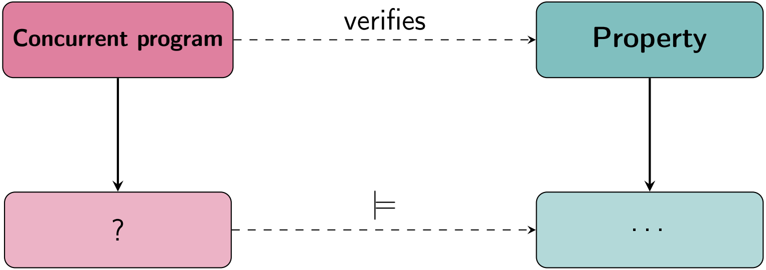
- We defined a new automaton model, called UPDS, that models the stack of a program more accurately than a PDS.
- We show that the backward and forward reachability sets of UPDSs are not regular, but that the latter is context-sensitive.
- We can either under-approximate or over-approximate these sets.
- We have shown some potential applications of this model.

Our third contribution:
Reachability analysis of synchronized pushdown networks

The model-checking framework



The model-checking framework



What about concurrent programs?

- Pushdown systems (PDSs) are a natural model for sequential programs.

Dynamic pushdown networks

- Pushdown systems (PDSs) are a natural model for sequential programs.
- Intuitively, one can model each thread of a program as a PDS. A concurrent program can therefore be seen as a network of PDSs.

Dynamic pushdown networks

- Pushdown systems (PDSs) are a natural model for sequential programs.
- Intuitively, one can model each thread of a program as a PDS. A concurrent program can therefore be seen as a network of PDSs.
- Hence, we consider dynamic pushdown network (DPN) model [Bouajjani, Müller-Olm, and Touili, CONCUR'05]. It is a network of PDSs where each member can perform internal actions and spawn other instances of PDSs.

A synchronization issue

However, in an actual parallel program, threads can communicate, but in a DPN, they can't.

We need therefore a more accurate model that can handle synchronization between threads. To this end, we extend DPNs with synchronization by rendez-vous.

Synchronization by rendez-vous

When two threads synchronize, one thread must send a signal a and the other, its co-signal $\overline{a}$.

$$\begin{array}{ccc} T_1 : c_1 & \dots & T_2 : c_2 \\ \downarrow a & \text{simultaneously} & \downarrow \overline{a} \\ T_1 : c_1' & \dots & T_2 : c_2' \end{array}$$

We define a set Act of actions that contains synchronization signals as well as an internal action τ.

Synchronized dynamic pushdown network

Definition

A synchronized dynamic pushdown network (SDPN) is a quadruplet $M = (Act, P, \Gamma, \Delta)$ where:

- P is a finite set of control states;
- Γ a finite stack alphabet disjoint from P;
- Δ a finite set of labelled transition rules featuring:
  - simple pushdown operations of the form $p\gamma \xrightarrow{l} p'w$, $l \in Act$;
  - thread spawns of the form $p\gamma \xrightarrow{l} p_2 w_2 \triangleright p_1 w_1$, $l \in Act$;

A configuration of a SDPN is a word in $(P\Gamma^*)^*$ that is a concatenation of all the configurations of the PDSs in the network.

The semantics: pushdown actions and spawns

If $p_1\gamma_1 \xrightarrow{l} p_1' w_1' \in \Delta$, then:

$$\ldots p_1\gamma_1 w_1 \ldots \xrightarrow{l}_M \ldots p_1' w_1' w_1 \ldots$$

The semantics: pushdown actions and spawns

If $p_1 \gamma_1 \xrightarrow{l} p'_1 w'_1 \in \Delta$, then:

$$\ldots p_1 \gamma_1 w_1 \ldots \xrightarrow{l}_M \ldots p'_1 w'_1 w_1 \ldots$$

If $p_1 \gamma_1 \xrightarrow{l} p_2 w_2 \triangleright p'_1 w'_1 \in \Delta$, then:

$$\ldots p_1 \gamma_1 w_1 \ldots \xrightarrow{l}_M \ldots p_2 w_2 p'_1 w'_1 w_1 \ldots$$

The semantics: pushdown actions and spawns

If $p_1\gamma_1 \xrightarrow{l} p_1' w_1' \in \Delta$, then:

$$\ldots p_1\gamma_1 w_1 \ldots \xrightarrow{l}_M \ldots p_1' w_1' w_1 \ldots$$

If $p_1\gamma_1 \xrightarrow{l} p_2 w_2 \triangleright p_1' w_1' \in \Delta$, then:

$$\ldots p_1\gamma_1 w_1 \ldots \xrightarrow{l}_M \ldots p_2 w_2 p_1' w_1' w_1 \ldots$$

l can be a signal a, a co-signal $\bar{a}$, or an internal action τ.

The semantics of synchronized transitions

If threads $p_1 \gamma_1 w_1$ and $p_2 \gamma_2 w_2$ can apply the pushdown rules $p_1 \gamma_1 \xrightarrow{a} p'_1 w'_1$ and $p_2 \gamma_2 \xrightarrow{\overline{a}} p'_2 w'_2 \in \Delta$:

$$\begin{array}{ccccc} \ldots & p_1 \gamma_1 w_1 & \ldots & p_2 \gamma_2 w_2 & \ldots \\ & \downarrow a & & \downarrow \overline{a} & \\ \ldots & p'_1 w'_1 w_1 & \ldots & p'_2 w'_2 w_2 & \ldots \end{array}$$

The semantics of synchronized transitions

If threads $p_1\gamma_1 w_1$ and $p_2\gamma_2 w_2$ can apply the pushdown rules $p_1\gamma_1 \xrightarrow{a} p_1' w_1'$ and $p_2\gamma_2 \xrightarrow{\overline{a}} p_2' w_2' \in \Delta$:

$$\begin{array}{ccccccc} \ldots & p_1\gamma_1 w_1 & \ldots & p_2\gamma_2 w_2 & \ldots \\ & \downarrow a & & \downarrow \overline{a} & \\ \ldots & p_1' w_1' w_1 & \ldots & p_2' w_2' w_2 & \ldots \end{array}$$

Then they can synchronize over the signal a:

$$\begin{array}{ccccccc} \ldots & p_1\gamma_1 w_1 & \ldots & p_2\gamma_2 w_2 & \ldots \\ & & \downarrow \tau & & \\ \ldots & p_1' w_1' w_1 & \ldots & p_2' w_2' w_2 & \ldots \end{array}$$

Invalid SDPN actions

In a real program, transitions of the form:

$$\ldots p_1 \gamma_1 w_1 \ldots \xrightarrow{l}_M \ldots p_1' w_1' w_1 \ldots$$

are only allowed if $l = \tau$ is an internal action.

Invalid SDPN actions

In a real program, transitions of the form:

$$\ldots p_1 \gamma_1 w_1 \ldots \xrightarrow{l}_M \ldots p_1' w_1' w_1 \ldots$$

are only allowed if $l = \tau$ is an internal action.

If $l = a$ or $l = \bar{a}$, then the program must wait for a matching synchronization action and the thread can't execute such a transition on its own.

Characterizing valid execution paths

As a consequence, a valid execution path in a program can only use internal transitions of the form:

$$\ldots p_1 \gamma_1 w_1 \ldots \xrightarrow{\tau}_M \ldots p_1' w_1' w_1 \ldots$$

Characterizing valid execution paths

As a consequence, a valid execution path in a program can only use internal transitions of the form:

$$\ldots p_1 \gamma_1 w_1 \ldots \xrightarrow{\tau}_M \ldots p_1' w_1' w_1 \ldots$$

Or, if $p_1 \gamma_1 \xrightarrow{a} p_1' w_1'$ and $p_2 \gamma_2 \xrightarrow{\bar{a}} p_2' w_2' \in \Delta$, synchronized transitions of the form:

$$\ldots p_1 \gamma_1 w_1 \ldots p_2 \gamma_2 w_2 \ldots \xrightarrow{\tau}_M \ldots p_1' w_1' w_1 \ldots p_2' w_2' w_2 \ldots$$

Characterizing valid execution paths

As a consequence, a valid execution path in a program can only use internal transitions of the form:

$$\ldots p_1 \gamma_1 w_1 \ldots \xrightarrow{\tau}_M \ldots p_1' w_1' w_1 \ldots$$

Or, if $p_1 \gamma_1 \xrightarrow{a} p_1' w_1'$ and $p_2 \gamma_2 \xrightarrow{\bar{a}} p_2' w_2' \in \Delta$, synchronized transitions of the form:

$$\ldots p_1 \gamma_1 w_1 \ldots p_2 \gamma_2 w_2 \ldots \xrightarrow{\tau}_M \ldots p_1' w_1' w_1 \ldots p_2' w_2' w_2 \ldots$$

Valid execution paths therefore only use transitions labelled by τ.

The reachability problem

Given a SDPN M and two regular sets of configuration C and C', we consider the reachability problem: is there a valid path of M leading from C to C'?

Given a SDPN M and two regular sets of configuration C and C', we consider the reachability problem: is there a valid path of M leading from C to C'?

This is equivalent to:

$$Paths(C, C') \cap \tau^* = \emptyset?$$

An undecidable problem

The reachability problem for synchronization-sensitive pushdown systems is undecidable [Ramalingam, '00], hence, for SDPNs as well. We cannot therefore compute $Paths(C, C')$.

An undecidable problem

The reachability problem for synchronization-sensitive pushdown systems is undecidable [Ramalingam, '00], hence, for SDPNs as well. We cannot therefore compute $Paths(C, C')$.

But if we consider an over-approximation:

$$\alpha(Paths(C, C')) \supseteq Paths(C, C')$$

Then:

$$\alpha(Paths(C, C')) \cap \tau^* = \emptyset$$

implies that:

$$Paths(C, C') \cap \tau^* = \emptyset$$

Characterizing $Paths(C, C')$

- We use finite-state automata A^C and $A^{C'}$ to represent the regular sets of configurations C and C'.

Characterizing $Paths(C, C')$

- We use finite-state automata A^C and $A^{C'}$ to represent the regular sets of configurations C and C'.
- By applying the saturation procedure of [Bouajjani et al., CONCUR'05] to $A^{C'}$, we compute a finite-state automaton A_{pre^*} accepting $pre^*(C')$.

Characterizing $Paths(C, C')$

- We use finite-state automata A^C and $A^{C'}$ to represent the regular sets of configurations C and C'.
- By applying the saturation procedure of [Bouajjani et al., CONCUR'05] to $A^{C'}$, we compute a finite-state automaton A_{pre^*} accepting $pre^*(C')$.
- We add extra labels $\lambda(t)$ to the edges t of the automaton A_{pre^*} in such a manner that the relabelled automaton A' accepts all the pairs (c, π) where $c \in pre^*(C')$ and $\pi = Paths(\{c\}, C')$.

Characterizing $Paths(C, C')$

- We use finite-state automata A^C and $A^{C'}$ to represent the regular sets of configurations C and C'.
- By applying the saturation procedure of [Bouajjani et al., CONCUR'05] to $A^{C'}$, we compute a finite-state automaton A_{pre^*} accepting $pre^*(C')$.
- We add extra labels $\lambda(t)$ to the edges t of the automaton A_{pre^*} in such a manner that the relabelled automaton A' accepts all the pairs (c, π) where $c \in pre^*(C')$ and $\pi = Paths(\{c\}, C')$.
- We consider the intersection of A' and A^C in order to compute $Paths(C, C')$.

Characterizing $Paths(C, C')$

- We use finite-state automata A^C and $A^{C'}$ to represent the regular sets of configurations C and C'.
- By applying the saturation procedure of [Bouajjani et al., CONCUR'05] to $A^{C'}$, we compute a finite-state automaton A_{pre^*} accepting $pre^*(C')$.
- We add extra labels $\lambda(t)$ to the edges t of the automaton A_{pre^*} in such a manner that the relabelled automaton A' accepts all the pairs (c, π) where $c \in pre^*(C')$ and $\pi = Paths(\{c\}, C')$.
- We consider the intersection of A' and A^C in order to compute $Paths(C, C')$.

Computing the labels

- Following [Bouajjani, Esparza, and Touili, POPL'03], we want to define a set of constraints whose least solution accurately characterizes the set of paths leading to C'.

Computing the labels

- Following [Bouajjani, Esparza, and Touili, POPL'03], we want to define a set of constraints whose least solution accurately characterizes the set of paths leading to C'.
- Unlike [Bouajjani, Esparza, and Touili, POPL'03], we cannot use sets of paths in 2^{Act^*} as labels because of the interleaving of executions paths of children threads in spawn rules.

Computing the labels

- Following [Bouajjani, Esparza, and Touili, POPL'03], we want to define a set of constraints whose least solution accurately characterizes the set of paths leading to C'.
- Unlike [Bouajjani, Esparza, and Touili, POPL'03], we cannot use sets of paths in 2^{Act^*} as labels because of the interleaving of executions paths of children threads in spawn rules.
- We show that, by using functions in $2^{Act^*} \to 2^{Act^*}$ as labels, we can accurately characterize the set of paths leading to C' with constraints.

Computing the labels

- Following [Bouajjani, Esparza, and Touili, POPL'03], we want to define a set of constraints whose least solution accurately characterizes the set of paths leading to C'.
- Unlike [Bouajjani, Esparza, and Touili, POPL'03], we cannot use sets of paths in 2^{Act^*} as labels because of the interleaving of executions paths of children threads in spawn rules.
- We show that, by using functions in $2^{Act^*} \to 2^{Act^*}$ as labels, we can accurately characterize the set of paths leading to C' with constraints.
- This set of constraints can't be solved because the reachability problem is the undecidable.

Computing the labels

- Following [Bouajjani, Esparza, and Touili, POPL'03], we want to define a set of constraints whose least solution accurately characterizes the set of paths leading to C'.
- Unlike [Bouajjani, Esparza, and Touili, POPL'03], we cannot use sets of paths in 2^{Act^*} as labels because of the interleaving of executions paths of children threads in spawn rules.
- We show that, by using functions in $2^{Act^*} \to 2^{Act^*}$ as labels, we can accurately characterize the set of paths leading to C' with constraints.
- This set of constraints can't be solved because the reachability problem is the undecidable.
- We therefore solve it in a finite abstract domain D in order to compute an over-approximation $\alpha(Paths(C, C'))$.

Finite-domain abstractions

We consider the domain $D = 2^W$, where W is the set of words of length smaller than n, and the n-th order prefix and suffix abstractions:

$$\textbf{Prefix: } \alpha_n^{prefix}(\{a_1 \dots a_n a_{n+1} \dots a_m\}) = \{a_1 \dots a_n\}$$

$$\textbf{Suffix: } \alpha_n^{suffix}(\{a_1 \dots a_{m-n} a_{m-n+1} \dots a_m\}) = \{a_{m-n+1} \dots a_m\}$$

An iterative abstraction scheme

1. We start from $n = 1$;

An iterative abstraction scheme

1. We start from $n = 1$;
2. we compute $\alpha(\mathit{Paths}(C, C'))$ for $\alpha = \alpha_n^{\mathit{prefix}}$ and $\alpha = \alpha_n^{\mathit{suffix}}$;

1. We start from $n = 1$;
2. we compute $\alpha(Paths(C, C'))$ for $\alpha = \alpha_n^{prefix}$ and $\alpha = \alpha_n^{suffix}$;
3. we check if $\alpha(Paths(C, C')) \cap \tau^* = \emptyset$; if this holds, then C' can't be reached from C with a valid path;

An iterative abstraction scheme

1. We start from $n = 1$;
2. we compute $\alpha(Paths(C, C'))$ for $\alpha = \alpha_n^{prefix}$ and $\alpha = \alpha_n^{suffix}$;
3. we check if $\alpha(Paths(C, C')) \cap \tau^* = \emptyset$; if this holds, then C' can't be reached from C with a valid path;
4. otherwise, we check if our abstraction introduced a spurious counter-example;

An iterative abstraction scheme

1. We start from $n = 1$;
2. we compute $\alpha(Paths(C, C'))$ for $\alpha = \alpha_n^{prefix}$ and $\alpha = \alpha_n^{suffix}$;
3. we check if $\alpha(Paths(C, C')) \cap \tau^* = \emptyset$; if this holds, then C' can't be reached from C with a valid path;
4. otherwise, we check if our abstraction introduced a spurious counter-example;
5. if the counter-example was spurious, we increment n and go back to the step 2.

We use this iterative abstraction scheme to find an error in a Bluetooth driver for Windows NT.

We model it as a SDPN and find that an erroneous configuration is reachable using a prefix abstraction of size 12.

Our third contribution

1. We defined a new model for concurrent programs called SDPN;

Our third contribution

1. We defined a new model for concurrent programs called SDPN;
2. We proved that its set of paths was the least solution of a set of constraints.

Our third contribution

1. We defined a new model for concurrent programs called SDPN;
2. We proved that its set of paths was the least solution of a set of constraints.
3. We used an abstraction framework in order to solve these constraints.

Our third contribution

1. We defined a new model for concurrent programs called SDPN;
2. We proved that its set of paths was the least solution of a set of constraints.
3. We used an abstraction framework in order to solve these constraints.
4. We over-approximated the reachability problem for SDPNs.

Our third contribution

1. We defined a new model for concurrent programs called SDPN;
2. We proved that its set of paths was the least solution of a set of constraints.
3. We used an abstraction framework in order to solve these constraints.
4. We over-approximated the reachability problem for SDPNs.
5. We defined an iterative abstraction scheme for SDPNs and applied it to a driver.

Conclusion

Summary

1. We showed that the model-checking problem of HyperLTL for PDSs is undecidable, we proved some decidability results when all variables are regular except the first, and we used these results to approximate the model-checking problem.

Summary

1. We showed that the model-checking problem of HyperLTL for PDSs is undecidable, we proved some decidability results when all variables are regular except the first, and we used these results to approximate the model-checking problem.
2. We defined a new automaton model, called UPDS, that models the stack of a program more accurately than a PDS, we proved that its reachability sets are not regular but context-sensitive, then we over and under-approximated these.

Summary

1. We showed that the model-checking problem of HyperLTL for PDSs is undecidable, we proved some decidability results when all variables are regular except the first, and we used these results to approximate the model-checking problem.
2. We defined a new automaton model, called UPDS, that models the stack of a program more accurately than a PDS, we proved that its reachability sets are not regular but context-sensitive, then we over and under-approximated these.
3. We defined a new model for concurrent programs called SDPN, we abstracted its reachability problem, and we applied this over-approximation in an iterative scheme.

Future work

1. We plan to implement algorithms to approximate the model-checking problem of HyperLTL for PDSs.

Future work

1. We plan to implement algorithms to approximate the model-checking problem of HyperLTL for PDSs.
2. We know that the reachability sets of the UPDS model are not regular; we want to determine whether they are context-free or not.

Future work

1. We plan to implement algorithms to approximate the model-checking problem of HyperLTL for PDSs.
2. We know that the reachability sets of the UPDS model are not regular; we want to determine whether they are context-free or not.
3. We plan to program a tool that would implement the abstraction framework for SDPNs designed in the third part of this thesis.

Thank you!

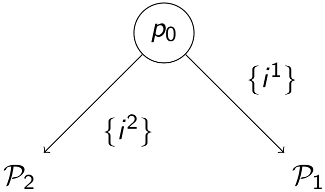
Publications

- Adrien Pommellet and Tayssir Touili, Model-checking HyperLTL for pushdown systems, 25th International Symposium on Model Checking of Software (SPIN'18).
- Adrien Pommellet, Marcio Diaz, and Tayssir Touili, Reachability analysis of pushdown systems with an upper stack, 11th International Conference on Language and Automata Theory and Applications (LATA'17).
- Adrien Pommellet and Tayssir Touili, Static analysis of multi-threaded recursive programs communicating via rendez-vous, 15th Asian Symposium on Programming Languages and Systems (APLAS'17).

Undecidability of HyperLTL model-checking

Proof of undecidability

Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two context-free languages accepted respectively by two PDA $\mathcal{P}_1$ and $\mathcal{P}_2$.



We design a PDS $\mathcal{P}$ that can simulate either $\mathcal{P}_1$ or $\mathcal{P}_2$, depending on its first transition.

Proof of undecidability

We want to design a HyperLTL formula on $\mathcal{P}$ such that it would characterize a common accepting run of $\mathcal{P}_1$ and $\mathcal{P}_2$.

$$\psi = \exists \pi_1, \exists \pi_2, \varphi$$

It will use two trace variables π_1 and π_2.

$$\psi = \exists \pi_1, \exists \pi_2, (i^1_{\pi_1} \wedge i^2_{\pi_2})$$

The trace variables π_1 and π_2 represent runs of $\mathcal{P}_1$ and $\mathcal{P}_2$ respectively.

$$\pi_1 : \{i^1\} \rightarrow \ldots$$
$$\pi_2 : \{i^2\} \rightarrow \ldots$$

Proof of undecidability

$$\psi = \exists \pi_1, \exists \pi_2, (i^1_{\pi_1} \wedge i^2_{\pi_2})$$

$$\wedge \ \mathrm{XG} \bigwedge_{a \in AP} (a_{\pi_1} \Leftrightarrow a_{\pi_2})$$

The two traces are equal from their second letter onwards.

$$\pi_1 : \{i^1\} \rightarrow \{a\} \rightarrow \dots$$

$$\pi_2 : \{i^2\} \rightarrow \{a\} \rightarrow \dots$$

Proof of undecidability

$$\psi = \exists \pi_1, \exists \pi_2, (i^1_{\pi_1} \wedge i^2_{\pi_2})$$

$$\wedge \; \mathrm{XG} \bigwedge_{a \in AP} (a_{\pi_1} \Leftrightarrow a_{\pi_2})$$

$$\wedge \; \mathrm{FG} \; (f_{\pi_1} \wedge f_{\pi_2})$$

The two traces are accepting.

$$\pi_1 : \{i^1\} \rightarrow \{a\} \rightarrow \ldots \rightarrow \{f\} \rightarrow \{f\} \rightarrow \ldots$$

$$\pi_2 : \{i^2\} \rightarrow \{a\} \rightarrow \ldots \rightarrow \{f\} \rightarrow \{f\} \rightarrow \ldots$$

Proof of undecidability

$\mathcal{P} \models \psi$ if and only if there is an accepting run π common to $\mathcal{P}_1$ and $\mathcal{P}_2$. But such a run exists if and only if $\mathcal{L}_1 \cap \mathcal{L}_2 \neq \emptyset$. $\square$

Hence:

Theorem

The model-checking problem of HyperLTL for pushdown systems is undecidable.

Non-regularity of ***post**** for UPDSs

A counter-example of regularity for *post**

We consider the UPDS $\mathcal{P}$:

$$\begin{array}{llll} (R_a) & (p, a) \rightarrow (p, \varepsilon) & (R_b) & (p, b) \rightarrow (p, \varepsilon) \\ (C) & (p, a) \rightarrow (p, ab) & & \end{array}$$

And the regular set $\mathcal{C} = \{p\} \times \{\varepsilon\} \times a\,(ba)^*$.

$$\mathbf{p}\ \boxed{a}\boxed{b}\boxed{a}\boxed{b}\boxed{a}$$

We consider the subset $L = \left\{ \left\langle p, a^{n+1}, b^n \right\rangle, n \in \mathbb{N} \right\} \subseteq \mathrm{post}^* \left(\mathcal{C} \right)$.

$$\mathbf{p}\ \boxed{a}\boxed{b}\boxed{a}\boxed{b}\boxed{a} \overset{R_a R_b}{\Longrightarrow} \boxed{a}\boxed{b}\ \mathbf{p}\ \boxed{a}\boxed{b}\boxed{a} \overset{C}{\Longrightarrow}$$

$$\boxed{a}\ \mathbf{p}\ \boxed{a}\boxed{b}\boxed{b}\boxed{a} \overset{R_a R_b R_b}{\Longrightarrow} \boxed{a}\boxed{a}\boxed{b}\boxed{b}\ \mathbf{p}\ \boxed{a} \overset{CC}{\Longrightarrow}$$

$$\boxed{a}\boxed{a}\ \mathbf{p}\ \boxed{a}\boxed{b}\boxed{b} \overset{R_a}{\Longrightarrow} \boxed{a}\boxed{a}\boxed{a}\ \mathbf{p}\ \boxed{b}\boxed{b} \in L$$

$$\begin{array}{llll} (R_a) & (p, a) \rightarrow (p, \varepsilon) & (R_b) & (p, b) \rightarrow (p, \varepsilon) \\ (C) & (p, a) \rightarrow (p, ab) & & \end{array}$$

A constraint on $post^*$

For any reachable configuration $\langle p, w_u, w_l \rangle$ and the word $w = \bar{w}_u w_l$, the inequality $|w|_b + |w|_{\bar{b}} + 1 \geq |w|_a + |w|_{\bar{a}}$ holds.

- The inequality holds on the starting configuration $\mathcal{C} = \{p\} \times \{\varepsilon\} \times a\,(ba)^*$.
- The rules $(R_a) = (p, a) \rightarrow (p, \varepsilon)$ and $(R_b) = (p, b) \rightarrow (p, \varepsilon)$ do not change the number of occurences of the letter a on the whole stack.
- The rule $(C) = (p, a) \rightarrow (p, ab)$ can make it smaller.

Applying the pumping lemma

If we suppose that $post^*(\mathcal{C})$ is regular, let k be its pumping length.

- We consider the word $w = \overline{a^{k+1}b^k}$ of the language L.
- We apply the pumping lemma to w: $w = xyz$, $|xy| \leq k$, $|y| \geq 1$, and $xy^iz \in post^*(\mathcal{C})$, $\forall i \geq 1$, with $x \in \bar{a}^*$, $y \in \bar{a}^+$ and $z \in \left(\bar{a} + \bar{b}\right)^*$.
- For i large enough, $w_i = xy^iz \in post^*(\mathcal{C})$ and $|w_i|_{\bar{a}} > |w_i|_{\bar{b}} + 1$.

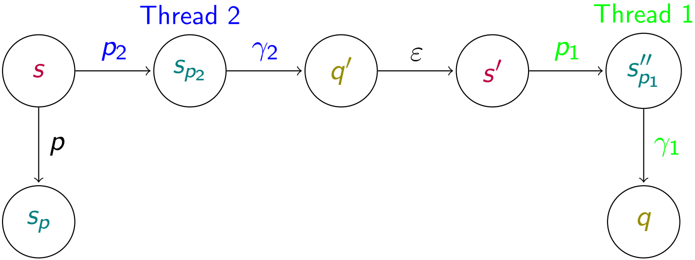
There is a contradiction and $post^*(\mathcal{C})$ is not regular.

Computing the constraints for SDPNs

There are five different types on constraints, depending on whether we are labelling a transition that was already in $A_{C'}$ or that was added by a saturation rule matched to a switch, a pop, a push, or a spawn.

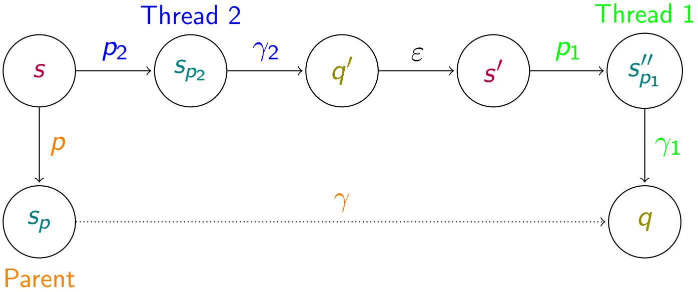
We focus on the latter spawn case.

For each rule $p\gamma \xrightarrow{a} p_2\gamma_2 \triangleright p_1\gamma_1 \in \Delta$:



The fifth constraint: spawn

For each rule $p\gamma \xrightarrow{a} p_2\gamma_2 \triangleright p_1\gamma_1 \in \Delta$:

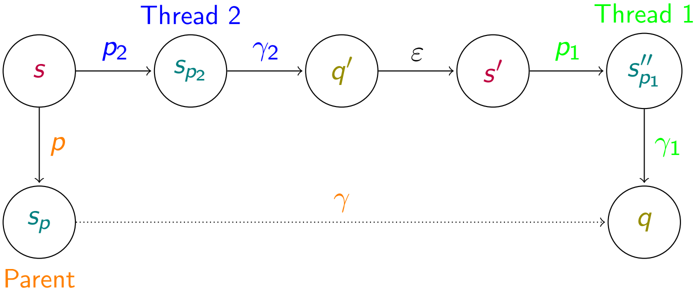


The fifth constraint: spawn

For each rule $p\gamma \xrightarrow{a} p_2\gamma_2 \triangleright p_1\gamma_1 \in \Delta$:

$$a \cdot (Paths(Thread_2) \sqcup Paths(Thread_1)) \subseteq Paths(Parent)$$

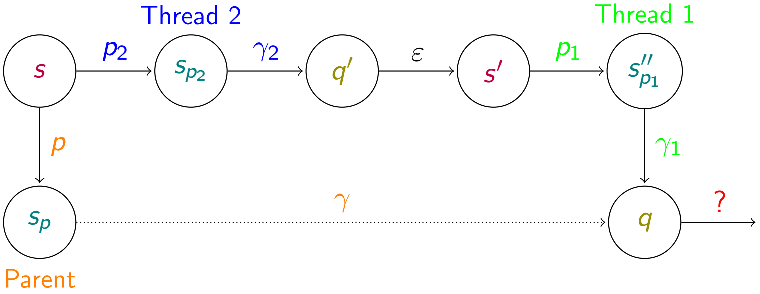
$$\longrightarrow a \cdot (\lambda(s_{p_2}, \gamma_2, q') \sqcup \lambda(s'_{p_1}, \gamma_1, q)) \subseteq \lambda(s_p, \gamma, q)$$



For each rule $p\gamma \xrightarrow{a} p_2\gamma_2 \triangleright p_1\gamma_1 \in \Delta$:

$$a \cdot (\mathit{Paths}(\mathit{Thread}_2) \sqcup \mathit{Paths}(\mathit{Thread}_1)) \subseteq \mathit{Paths}(\mathit{Parent})$$

$$\longrightarrow a \cdot (\lambda(s_{p_2}, \gamma_2, q') \sqcup \cancel{\lambda(s'_{p_1}, \gamma_1, q)}) \subseteq \lambda(s_p, \gamma, q)$$

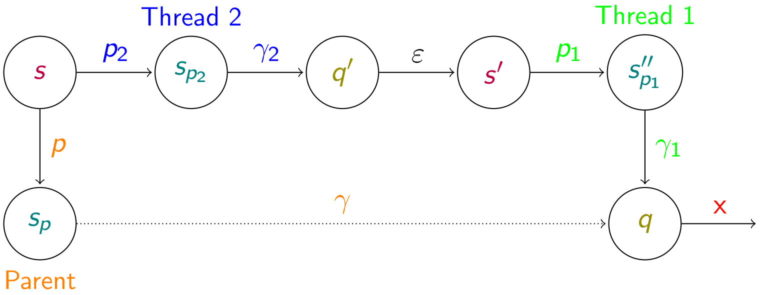


$\lambda(s'_{p_1}, \gamma_1, q)$ does not fully express $\mathit{Paths}(\mathit{Thread}_1)$!

For each rule $p\gamma \xrightarrow{a} p_2\gamma_2 \triangleright p_1\gamma_1 \in \Delta$:

$$a \cdot (Paths(Thread_2) \sqcup Paths(Thread_1)) \subseteq Paths(Parent)$$

$$\longrightarrow a \cdot (\lambda(s_{p_2}, \gamma_2, q') \sqcup \lambda(s'_{p_1}, \gamma_1, q)(x)) \subseteq \lambda(s_p, \gamma, q)(x)$$



We use functions in $\Pi \longrightarrow \Pi$ with a variable x.

Five constraints

$$\textbf{Initial:}\;\; Id \subseteq \lambda(t)$$

$$\textbf{Switch:}\;\; a \cdot \lambda(s_{p'}, \gamma', q)(x) \subseteq \lambda(s_p, \gamma, q)(x)$$

$$\textbf{Pop:}\;\; \{a\} \subseteq \lambda(s_p, \gamma, s_{p'})(x)$$

$$\textbf{Push:}\;\; a \cdot (\lambda(s_{p'}, \gamma_1, q') \circ \lambda(q', \gamma_2, q)(x)) \subseteq \lambda(s_p, \gamma, q)(x)$$

$$\textbf{Spawn:}\;\; a \cdot (\lambda(s_{p_2}, \gamma_2, q')(\{\varepsilon\}) \sqcup \lambda(s'_{p_1}, \gamma_1, q)(x)) \subseteq \lambda(s_p, \gamma, q)(x)$$