

Verifying List-Manipulating Programs with Unbounded Thread Creation

Thomas Noll Stefan Rieger

MOVES: Software Modeling and Verification
RWTH Aachen University, Germany

27.08.2007



FR 6.2 Informatik, Universität des Saarlandes, Saarbrücken

Overview

- ① Programming Language and Logic
- ② Data Abstraction
- ③ Control-Flow Model
- ④ Model Checking
- ⑤ Conclusion

Aim: Verification of dynamic data structures

Problem

- unbounded creation and destruction of **objects** at runtime
- unbounded creation of **threads** at runtime
- destructive updates via pointers
- parallelism
- possibly **infinite state space**

Aim: Verification of dynamic data structures

Problem

- unbounded creation and destruction of **objects** at runtime
- unbounded creation of **threads** at runtime
- destructive updates via pointers
- parallelism
- possibly **infinite state space**

Approach

- simple pointer programming language
- temporal pointer logic
- heap **abstraction**
- **at most one outgoing edge (list-like structures)**

A List Manipulating Language

DLM-Program

For program variables $v_i, v \in PV$ and process names $p_j, p \in \mathcal{P}$:

$$\pi = \mathbf{var} \ v_1, \dots, v_k; \ \mathbf{proc} \ \mathbf{main}(s_0); \ p_1(s_1); \dots; \ p_l(s_l)$$

A List Manipulating Language

DLM-Program

For program variables $v_i, v \in PV$ and process names $p_j, p \in \mathcal{P}$:

$$\pi = \mathbf{var} \ v_1, \dots, v_k; \ \mathbf{proc} \ \mathbf{main}(s_0); \ p_1(s_1); \dots; \ p_l(s_l)$$

Statements ($s_i = s_{i1}; \dots; s_{ir_i}$)

signal

PExp := PExp

if BExp **goto** n

goto n

atc(BExp)

end atc

new(PExp)

del(PExp)

spawn(p)

exit

A List Manipulating Language

DLM-Program

For program variables $v_i, v \in PV$ and process names $p_j, p \in \mathcal{P}$:

$$\pi = \mathbf{var} \ v_1, \dots, v_k; \ \mathbf{proc} \ \mathbf{main}(s_0); \ p_1(s_1); \dots; \ p_l(s_l)$$

Statements ($s_i = s_{i1}; \dots; s_{ir_i}$)

signal

PExp := PExp

if BExp **goto** n

goto n

atc(BExp)

end atc

new(PExp)

del(PExp)

spawn(p)

exit

Pointer Expressions

$$\text{PExp} ::= \mathit{nil} \mid v \mid *v \mid \&v$$

main

```
var x, y;  
proc main(  
01  new(x);  
02  spawn(server);  
)
```

main

```
var  $x, y$ ;  
proc main(  
01  new( $x$ );  
02  spawn(server);  
)
```

server

```
server(  
11  spawn(worker);  
12  atc(tt);  
13      $y := x$ ;  
14     new( $x$ );  
15      $*x := y$ ;  
16  end atc;  
17  goto 1;  
)
```

main

```
var  $x, y$ ;  
proc main(  
01  new( $x$ );  
02  spawn(server);  
)
```

server

```
server(  
11  spawn(worker);  
12  atc(tt);  
13      $y := x$ ;  
14     new( $x$ );  
15      $*x := y$ ;  
16  end atc;  
17  goto 1;  
)
```

worker

```
worker(  
21  atc( $x \neq nil$ );  
22      $y := x$ ;  
23      $x := *x$ ;  
24     del( $y$ );  
25  end atc;  
)
```

main

```

var x, y;
proc main(
01  new(x);
02  spawn(server);
)

```

server

```

server(
11  spawn(worker);
12  atc(tt);
13     y := x;
14     new(x);
15     *x := y;
16  end atc;
17  goto 1;
)

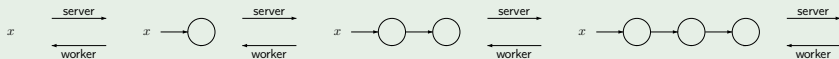
```

worker

```

worker(
21  atc(x ≠ nil);
22     y := x;
23     x := *x;
24     del(y);
25  end atc;
)

```



A Heap Logic

Pointer Logic (PL)

$$\text{NExp} ::= \text{nil} \mid v (\in PV) \mid x (\in LV) \mid * \text{NExp}$$

A Heap Logic

Pointer Logic (PL)

$\text{NExp} ::= nil \mid v (\in PV) \mid x (\in LV) \mid * \text{NExp}$
 $\text{Atomic} ::= tt \mid ff \mid \text{new}_n \mid \text{del} \mid \text{err} \mid \text{leak} \mid \text{spawn}_p$
 $\quad \mid \text{NExp} = \text{NExp} \mid \text{NExp} \rightsquigarrow \text{NExp}$

A Heap Logic

Pointer Logic (PL)

$$\begin{aligned} \text{NExp} &::= \text{nil} \mid v (\in PV) \mid x (\in LV) \mid * \text{NExp} \\ \text{Atomic} &::= \text{tt} \mid \text{ff} \mid \text{new}_n \mid \text{del} \mid \text{err} \mid \text{leak} \mid \text{spawn}_p \\ &\quad \mid \text{NExp} = \text{NExp} \mid \text{NExp} \rightsquigarrow \text{NExp} \\ \text{PL} &::= \text{Atomic} \mid \neg \text{PL} \mid \text{PL} \wedge \text{PL} \mid \exists x : \text{PL} \end{aligned}$$

A Heap Logic

Pointer Logic (PL)

$$\begin{aligned}
 \text{NExp} & ::= \text{nil} \mid v (\in PV) \mid x (\in LV) \mid * \text{NExp} \\
 \text{Atomic} & ::= \text{tt} \mid \text{ff} \mid \text{new}_n \mid \text{del} \mid \text{err} \mid \text{leak} \mid \text{spawn}_p \\
 & \quad \mid \text{NExp} = \text{NExp} \mid \text{NExp} \rightsquigarrow \text{NExp} \\
 \text{PL} & ::= \text{Atomic} \mid \neg \text{PL} \mid \text{PL} \wedge \text{PL} \mid \exists x : \text{PL}
 \end{aligned}$$

Temporal Pointer Logic (TPL, finite traces allowed)

$$\text{TPL} ::= \text{PL} \mid \neg \text{TPL} \mid \text{TPL} \wedge \text{TPL} \mid \mathbf{X} \text{TPL} \mid \text{TPL} \mathbf{U} \text{TPL}$$

A Heap Logic

Pointer Logic (PL)

$$\begin{aligned}
 \text{NExp} & ::= \text{nil} \mid v (\in PV) \mid x (\in LV) \mid * \text{NExp} \\
 \text{Atomic} & ::= \text{tt} \mid \text{ff} \mid \text{new}_n \mid \text{del} \mid \text{err} \mid \text{leak} \mid \text{spawn}_p \\
 & \quad \mid \text{NExp} = \text{NExp} \mid \text{NExp} \rightsquigarrow \text{NExp} \\
 \text{PL} & ::= \text{Atomic} \mid \neg \text{PL} \mid \text{PL} \wedge \text{PL} \mid \exists x : \text{PL}
 \end{aligned}$$

Temporal Pointer Logic (TPL, finite traces allowed)

$$\text{TPL} ::= \text{PL} \mid \neg \text{TPL} \mid \text{TPL} \wedge \text{TPL} \mid \mathbf{X} \text{TPL} \mid \text{TPL} \mathbf{U} \text{TPL}$$

Abbreviations

$$\mathbf{A}\varphi := \neg \mathbf{E} \neg \varphi \quad \mathbf{F}\psi := \text{tt} \mathbf{U} \psi \quad \mathbf{G}\psi := \neg \mathbf{F} \neg \psi$$

main

```
var  $x, y$ ;  
proc main(  
01  new( $x$ );  
02  spawn(server); )
```

server

```
server(  
11  spawn(worker);  
12  atc(tt);  
13     $y := x$ ;  
14    new( $x$ );  
15     $*x := y$ ;  
16  end atc;  
17  goto 1; )
```

worker

```
worker(  
21  atc( $x \neq nil$ );  
22     $y := x$ ;  
23     $x := *x$ ;  
24    del( $y$ );  
25  end atc; )
```

Properties

1. $(\mathbf{GX} \text{ tt}) \wedge (\neg \mathbf{F} \text{ err})$
2. $\mathbf{GF} \exists n : \text{new}_n$
3. $\mathbf{GF} \text{spawn}_{\text{worker}}$
4. $\mathbf{G}(\exists n : \text{new}_n \rightarrow \mathbf{F} \text{spawn}_{\text{worker}})$
5. $\neg \mathbf{G}(\text{spawn}_{\text{worker}} \rightarrow \mathbf{F} \text{del})$

Expressibility

Limitation

- no temporal operators within quantifiers
- ⇒ no liveness properties like $\forall x : (\text{new } x \rightarrow (\mathbf{F} \text{ del } x))$

Expressibility

Limitation

- no temporal operators within quantifiers
- ⇒ no liveness properties like $\forall x : (\text{new } x \rightarrow (\mathbf{F} \text{ del } x))$

Advantage

- no binding of logical variables across states necessary
- reduction to **standard LTL model checking** (with finite traces)

Overview

- ① Programming Language and Logic
- ② Data Abstraction
- ③ Control-Flow Model
- ④ Model Checking
- ⑤ Conclusion

Idea

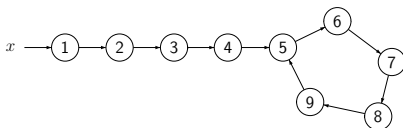
Global precision constant M

- M parametrizes the whole abstraction
- M depends on the formula to verify (lower bound)

Idea

Global precision constant M

- M parametrizes the whole abstraction
- M depends on the formula to verify (lower bound)



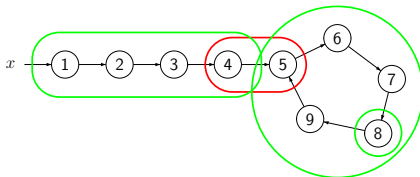
Approach

- Merge nodes along **chains** into a single **summary node**.

Idea

Global precision constant M

- M parametrizes the whole abstraction
- M depends on the formula to verify (lower bound)



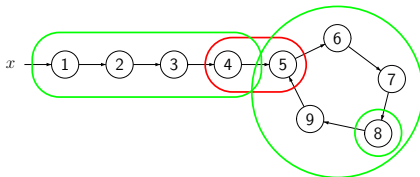
Approach

- Merge nodes along **chains** into a single **summary node**.

Idea

Global precision constant M

- M parametrizes the whole abstraction
- M depends on the formula to verify (lower bound)



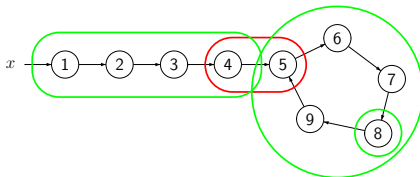
Approach

- Merge nodes along **chains** into a single **summary node**.
- Do only abstract from chains with $> M$ nodes.

Idea

Global precision constant M

- M parametrizes the whole abstraction
- M depends on the formula to verify (lower bound)



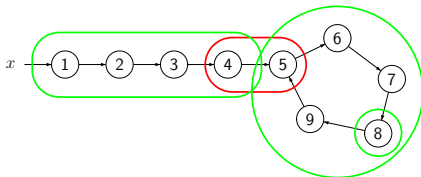
Approach

- Merge nodes along **chains** into a single **summary node**.
- Do only abstract from chains with $> M$ nodes.
- Define a **normal form** (uniqueness).

Idea

Global precision constant M

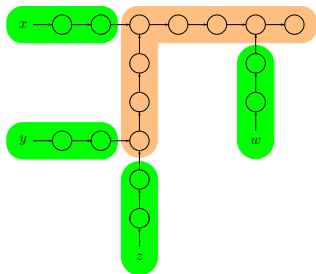
- M parametrizes the whole abstraction
- M depends on the formula to verify (lower bound)



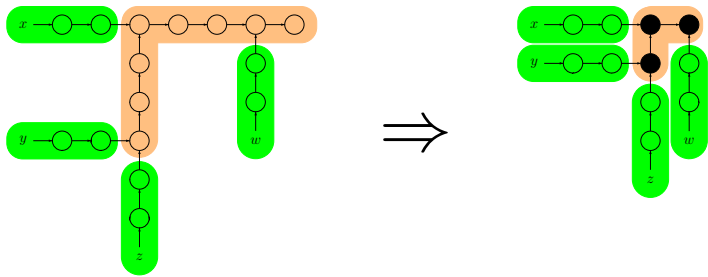
Approach

- Merge nodes along **chains** into a single **summary node**.
- Do only abstract from chains with $> M$ nodes.
- Define a **normal form** (uniqueness).
- Automatic **garbage collection** (finiteness) (\Rightarrow **leak-flag**).

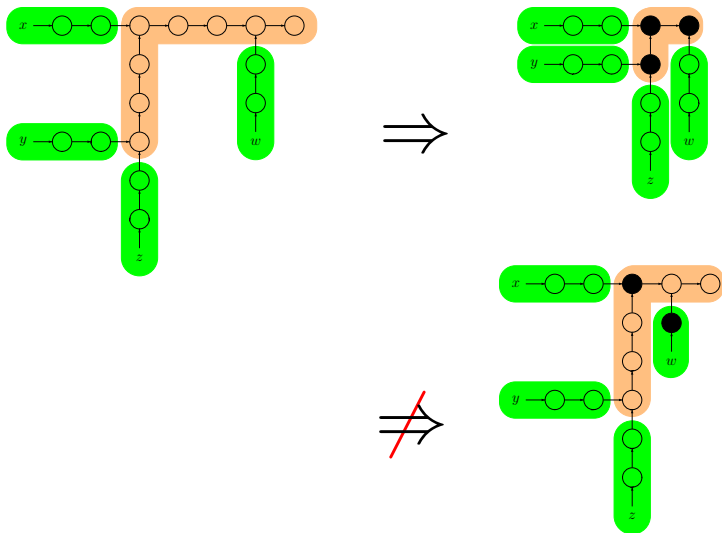
Canonical Representation (Example: $M = 1$)



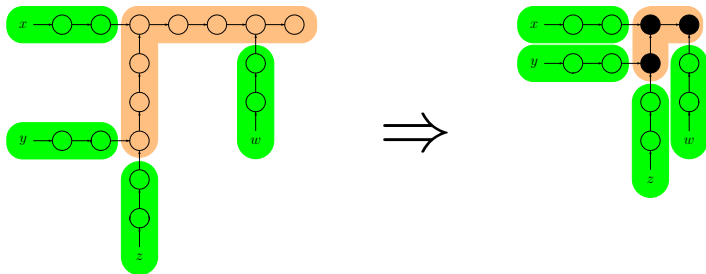
Canonical Representation (Example: $M = 1$)



Canonical Representation (Example: $M = 1$)



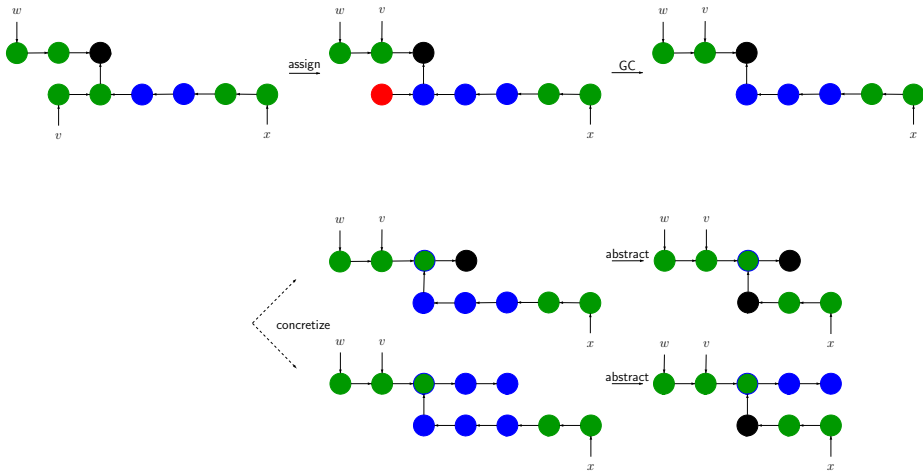
Canonical Representation (Example: $M = 1$)



Theorem

For every concrete heap configuration it **exists** a **unique** canonical configuration that is related by an abstraction morphism.

Example: Assignment $v := *w$ ($M = 2$)



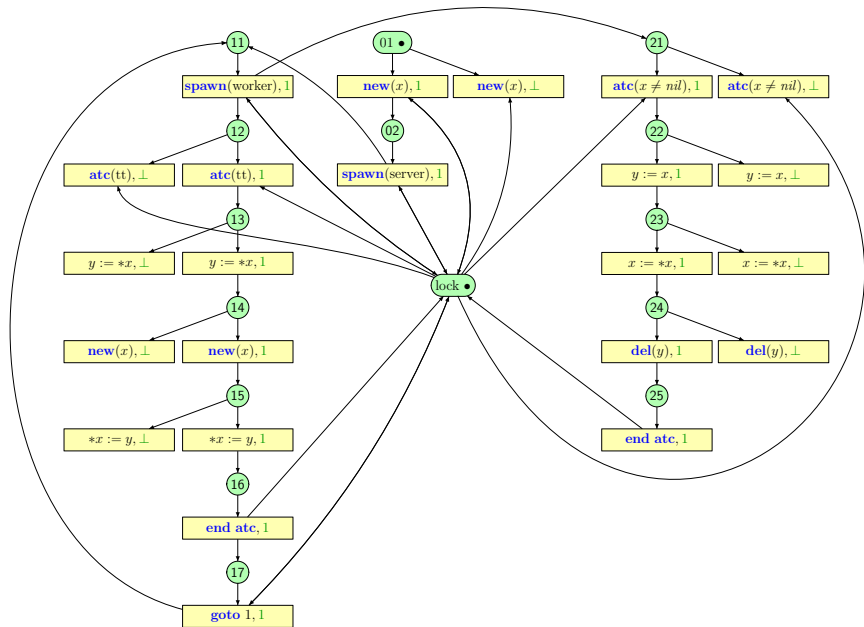
Overview

- ① Programming Language and Logic
- ② Data Abstraction
- ③ Control-Flow Model**
- ④ Model Checking
- ⑤ Conclusion

Idea

Modeling Control-Flow by Petri net \mathfrak{P}^c

- Every control location corresponds to a place
- Transitions are labeled by instruction and flag $\in \{0, 1, \perp\}$
- Place **lock** for synchronization



Combining Control-Flow and Heap Semantics

Heap Semantics

Interpret abstract heap semantics as 1-safe Petri net \mathfrak{P}^h whose transitions are labeled by instruction and $\text{flag} \in \{0, 1, \perp\}$

Combining Control-Flow and Heap Semantics

Heap Semantics

Interpret abstract heap semantics as 1-safe Petri net \mathfrak{P}^h whose transitions are labeled by instruction and flag $\in \{0, 1, \perp\}$

Data-Abstract Program Semantics

Given: $\mathfrak{P}^c = (P, T, \ell, m_0)$ and $\mathfrak{P}^h = (P, T, \ell, m_0)$

Define: $\mathfrak{P} := \mathfrak{P}^c \times \mathfrak{P}^h := (P \cup P, T, \ell, m_0)$

$$T = \{(\bullet t \cup \bullet t, t \bullet \cup t \bullet) \mid t \in T \wedge t \in T \wedge \ell(t) = \ell(t)\}$$

$$m_0(p) = \begin{cases} m_0(p) & \text{if } p \in P \\ m_0(p) & \text{otherwise} \end{cases}$$

Combining Control-Flow and Heap Semantics

Heap Semantics

Interpret abstract heap semantics as 1-safe Petri net \mathfrak{P}^h whose transitions are labeled by instruction and $\text{flag} \in \{0, 1, \perp\}$

Data-Abstract Program Semantics

Given: $\mathfrak{P}^c = (P, T, \ell, m_0)$ and $\mathfrak{P}^h = (P, T, \ell, m_0)$

Define: $\mathfrak{P} := \mathfrak{P}^c \times \mathfrak{P}^h := (P \cup P, T, \ell, m_0)$

$$T = \{(\bullet t \cup \bullet t, t^\bullet \cup t^\bullet) \mid t \in T \wedge t \in T \wedge \ell(t) = \ell(t)\}$$

$$m_0(p) = \begin{cases} m_0(p) & \text{if } p \in P \\ m_0(p) & \text{otherwise} \end{cases}$$

Observation

\mathfrak{P} is **unbounded** and cannot be represented by a finite LTS.

Overview

- ① Programming Language and Logic
- ② Data Abstraction
- ③ Control-Flow Model
- ④ Model Checking
- ⑤ Conclusion

Semantics of Pointer Logic

Remark

Semantics of Pointer Logic on [abstract states](#) needs complex machinery \Rightarrow here omitted

Semantics of Pointer Logic

Remark

Semantics of Pointer Logic on **abstract states** needs complex machinery \Rightarrow here omitted

Proposition (Correctness)

$\mathbf{H} \models \varphi$ for an abstract heap \mathbf{H} if and only if $H \models \varphi$ for all concrete heaps H represented by \mathbf{H} .

Semantics of Pointer Logic

Remark

Semantics of Pointer Logic on **abstract states** needs complex machinery \Rightarrow here omitted

Proposition (Correctness)

$\mathbf{H} \models \varphi$ for an abstract heap \mathbf{H} if and only if $H \models \varphi$ for all concrete heaps H represented by \mathbf{H} .

Note

To fulfill the proposition, the abstraction precision constant M is chosen **dependent on φ** .

TPL \rightarrow LTL

Evaluation of Subformulas

- Let $\varphi \in \text{TPL}$ and $\psi_1, \dots, \psi_n \in \text{PL}$ be its **maximal** PL subformulae.
- Every ψ_i is identified with an **atomic proposition** a_i
- The evaluation of ψ_1, \dots, ψ_n on a heap H thus yields a set of atomic propositions A_H ($a_i \in A_H \Leftrightarrow H \models \psi_i$).

TPL \rightarrow LTL

Evaluation of Subformulas

- Let $\varphi \in \text{TPL}$ and $\psi_1, \dots, \psi_n \in \text{PL}$ be its **maximal** PL subformulae.
- Every ψ_i is identified with an **atomic proposition** a_i
- The evaluation of ψ_1, \dots, ψ_n on a heap H thus yields a set of atomic propositions A_H ($a_i \in A_H \Leftrightarrow H \models \psi_i$).

Adapted Petri Net

Let $\mathfrak{P} = (P, T, \ell, m_0)$. Then define $\mathfrak{P}' = (P, T, \ell', m_0)$ as follows:

$$\ell'(t) = A_H \Leftrightarrow H \in t^\bullet$$

TPL \rightarrow LTL

Evaluation of Subformulas

- Let $\varphi \in \text{TPL}$ and $\psi_1, \dots, \psi_n \in \text{PL}$ be its **maximal** PL subformulae.
- Every ψ_i is identified with an **atomic proposition** a_i
- The evaluation of ψ_1, \dots, ψ_n on a heap H thus yields a set of atomic propositions A_H ($a_i \in A_H \Leftrightarrow H \models \psi_i$).

Adapted Petri Net

Let $\mathfrak{P} = (P, T, \ell, m_0)$. Then define $\mathfrak{P}' = (P, T, \ell', m_0)$ as follows:

$$\ell'(t) = A_H \Leftrightarrow H \in t^\bullet$$

LTL Formula

$\varphi' = \varphi[\psi_1/a_1, \dots, \psi_n/a_n]$ is an **LTL formula**

First Result

Theorem

It is **decidable** whether $\mathfrak{P}' \models \varphi'$.

First Result

Theorem

It is **decidable** whether $\mathfrak{P}' \models \varphi'$.

Proof

Reduction to the **reachability problem** for Petri nets

Exponentially many instances, each solvable in EXPSPACE.

First Result

Theorem

It is **decidable** whether $\mathfrak{P}' \models \varphi'$.

Proof

Reduction to the **reachability problem** for Petri nets
Exponentially many instances, each solvable in EXPSPACE.

Consequence

To obtain a practically applicable algorithm, further abstraction is necessary.

Enforcing Boundedness

Abstract Petri Net

- Markings of the form $m : P \rightarrow \mathbb{C}$, where $\mathbb{C} = \{0, \dots, C, \star\}$
- Represent all values $> C$ by \star
- Introduction of **nondeterminism**

Enforcing Boundedness

Abstract Petri Net

- Markings of the form $m : P \rightarrow \mathbb{C}$, where $\mathbb{C} = \{0, \dots, C, \star\}$
- Represent all values $> C$ by \star
- Introduction of **nondeterminism**

Result

The **abstract** Petri net \mathfrak{P}' can equivalently be represented by a finite LTS.

Enforcing Boundedness

Abstract Petri Net

- Markings of the form $m : P \rightarrow \mathbb{C}$, where $\mathbb{C} = \{0, \dots, C, \star\}$
- Represent all values $> C$ by \star
- Introduction of **nondeterminism**

Result

The **abstract** Petri net \mathfrak{P}' can equivalently be represented by a finite LTS.

Model Checking

Generate the LTS T corresponding to \mathfrak{P}' and apply an LTL model checking algorithm^a to verify $T \models \varphi$. If $T \models \varphi$ then also $\pi \models \varphi$.

^aLTL with finite traces

Overview

- ① Programming Language and Logic
- ② Data Abstraction
- ③ Control-Flow Model
- ④ Model Checking
- ⑤ Conclusion

Summary

- **List-manipulating** programming language with
 - dynamic memory allocation
 - dynamic threading
 - destructive updates
- Data and control-flow abstraction handled **separately**
- Reduction to **standard LTL model checking** (on finite traces)
- **MC problem decidable for data-abstract semantics**
- Refinement via **global precision parameters**

Summary

- **List-manipulating** programming language with
 - dynamic memory allocation
 - dynamic threading
 - destructive updates
- Data and control-flow abstraction handled **separately**
- Reduction to **standard LTL model checking** (on finite traces)
- **MC problem decidable for data-abstract semantics**
- Refinement via **global precision parameters**

Outlook

- Integrating **data-values** (e.g. integers)
- Handling **thread-local variables**
- **Refinement** based on counterexamples
- **Extension towards more general data structures**