State Space Reduction for Sensor Networks using Two-level Partial Order Reduction

Manchun Zheng[1], David Sanán[2], Jun Sun[1], Yang Liu[3], Jin Song Dong[4], Yu Gu[1]

[1]Singapore University of Technology and Design

[2]Trinity College Dublin

[3]Nanyang Technology University

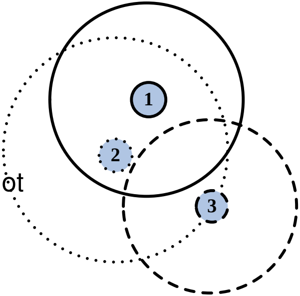
[4]National University of Singapore

VMCAI 2013

Outline

Sensor Networks (SNs)

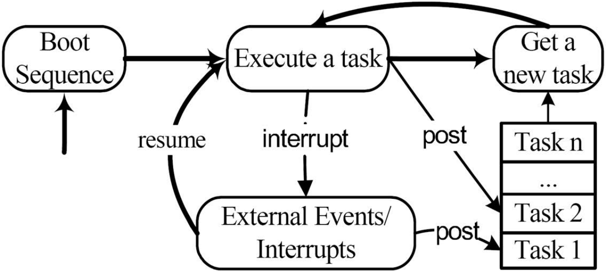
- Wireless communication: unicast, broadcast, dissemination, etc.
- Interrupt-driven behaviors.
- Hardware device: light, temperature, movement, etc.
- Applications:
  - Transportation: railway signaling
  - Military: enemy intrusion detection, autopilot
  - Environment: fire detection, landslide detection
- Reliability is important.



Sensor Network Programs

TinyOS: a lightweight operating system [LMP+04]

- Designed to run on small, wireless sensors
- Concurrent, interrupt-driven execution model
- Component libraries for device-related operations



Sensor Network Programs

TinyOS: a lightweight operating system [LMP+04]

- Designed to run on small, wireless sensors
- Concurrent, interrupt-driven execution model
- Component libraries for device-related operations

NesC (Networked Embedded System C) [GLvB+03]

- A dialect of C
- Component-based programming model
- Extension for concepts like command, event, tasks, etc
- Operations are split-phase

Formal Definitions of SNs [ZSL⁺11]

- A sensor network $\mathcal{N}$ is defined as $\{\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_n\}$ where $\mathcal{S}_i (0 \leqslant i \leqslant n)$ is a sensor with the identity i (i.e., the i^{th} sensor);
- Local state: a state C for a sensor $\mathcal{S}$ is (V, Q, B, P) where V is the valuation of variables, Q is the task queue, B is the message buffer and P is the program counter;
- Global state: a state $\mathcal{C}$ for a network $\mathcal{N}$ (global state) is $\{C_1, C_2, \cdots, C_n\}$, where $C_i (1 \leqslant i \leqslant n)$ is the i^{th} sensor's state.

Model Checking SN Implementation

- Model checkers: T-Check [LR10], Anquiro [MVO$^+$10], Tos2CProver [BK10]
- Limitations:
  - few deals with the (equivalently) complete state space
    - adopts stateless model checking techniques [LR10]
    - applies "abstraction" that ignores certain behavior [MVO$^+$10]
  - few deals with liveness properties but only safety properties
  - few reduction techniques are explored
    - only deals with communication event pairs [LR10]

Two-level Concurrency of SNs

- Network level: interleaving among different sensors

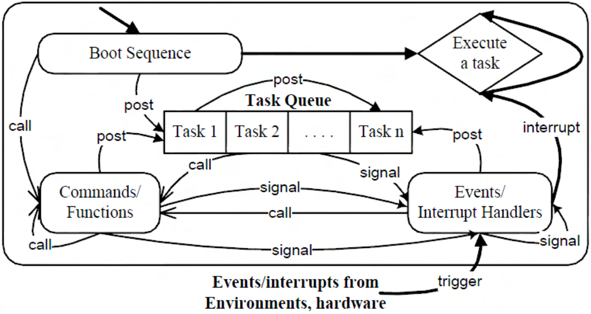
$$\frac{C_i \xrightarrow{e} C_i',\ e \neq s_i.send.dst.msg,\ e \neq s_i.idle,\ e \neq s_i.stop}{\{C_0, \cdots, C_i, \cdots, C_n\} \stackrel{e}{\hookrightarrow} \{C_0, \cdots, C_i', \cdots, C_n\}} \quad [\ nw1\ ]$$

Two-level Concurrency of SNs

- Network level: interleaving among different sensors

$$\frac{C_i \xrightarrow{e} C_i', \; e \neq s_i.send.dst.msg, \; e \neq s_i.idle, \; e \neq s_i.stop}{\{C_0, \cdots, C_i, \cdots, C_n\} \xhookrightarrow{e} \{C_0, \cdots, C_i', \cdots, C_n\}} \; [\, nw1 \,]$$

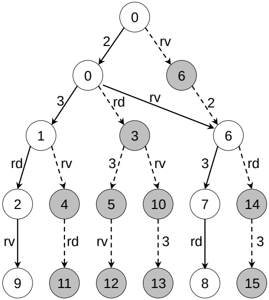
- Sensor level: concurrency of tasks and interrupt handlers



Motivating Example

```
event void Boot.booted(){
    call Read.read();
    post send_task();
}
event void Read.rdDone(int v){
    value += v;
}
task void send_task(){
    busy = TRUE;
    call Send.send(count);
}
event void Send.sendDone(){
    busy = FALSE;
}
event void Receive.receive(){
    count++;
    post send_task();
}
```

(a) Example Code



0: empty
1: t_{st}
2: t_{st}, t_{rd}
3: t_{rd}
4: t_{st}, t_{rv}
5: t_{rd}, t_{st}
6: t_{rv}
7: t_{rv}, t_{st}
8: t_{rv}, t_{st}, t_{rd}
9: t_{st}, t_{rd}, t_{rv}
10: t_{rd}, t_{rv}
11: t_{st}, t_{rv}, t_{rd}
12: t_{rd}, t_{st}, t_{rv}
13: t_{rd}, t_{rv}, t_{st}
14: t_{rv}, t_{rd}
15: t_{rv}, t_{rd}, t_{st}

(b) State space of Boot.booted

Motivating Example

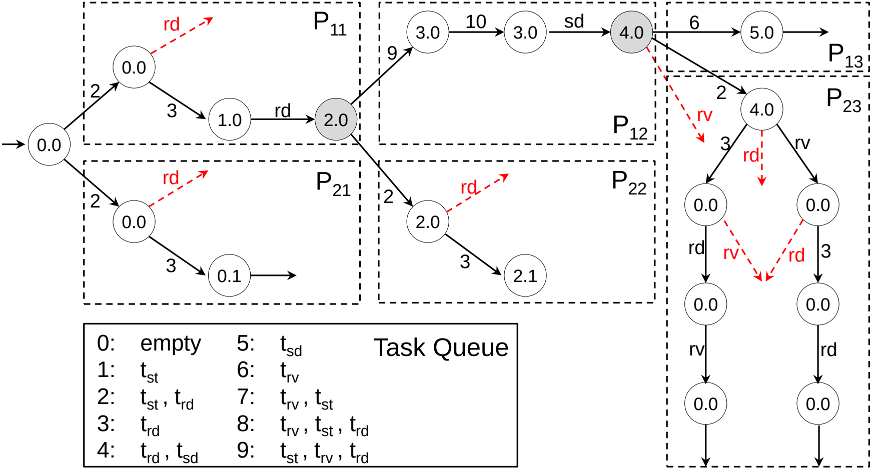


Figure: State space of two sensors running Boot.booted

Two-level POR

Overview

Exploring the state space by the SN Cartesian semantics

- Static analysis to identify w.r.t to a given property φ
  - local independence (to reduce intra-sensor concurrency)
    - variable access conflicts
    - φ-visible variable assignment
    - task queue equivalence
  - global independence (to reduce inter-sensor concurrency)
    - sending a message updates some sensors' message buffer and
    - triggers a receive interrupts that eventually modifies the task queue
- Reduction for model checking
  - establishing a smaller state space but preserving φ
  - applying on-the-fly model checking

Definition (Local Independence)

Given a state C, $\alpha_1, \alpha_2 \in \sum$, and $\alpha_1, \alpha_2 \in \mathit{enable}(C)$, actions α_1 and α_2 are said to be local-independent, denoted by $\alpha_1 \equiv_{LI} \alpha_2$, if the following conditions are satisfied.

1. $ex(ex(C, \alpha_1), \alpha_2) =_v ex(ex(C, \alpha_2), \alpha_1)$;
2. $Q(ex(ex(C, \alpha_1), \alpha_2)) \simeq Q(ex(ex(C, \alpha_2)), \alpha_1)$.

Local effects of an action

- updating an variable or enqueuing a task

Equivalent execution sequences when two actions

- access variables exclusively
- resulting in equivalent task queues

Definition (Local Independence)

Given a state C, $\alpha_1, \alpha_2 \in \sum$, and $\alpha_1, \alpha_2 \in enable(C)$, actions α_1 and α_2 are said to be local-independent, denoted by $\alpha_1 \equiv_{LI} \alpha_2$, if the following conditions are satisfied.

1. $ex(ex(C, \alpha_1), \alpha_2) =_v ex(ex(C, \alpha_2), \alpha_1)$;
2. $Q(ex(ex(C, \alpha_1), \alpha_2)) \simeq Q(ex(ex(C, \alpha_2)), \alpha_1)$.

Local effects of an action

- updating an variable or enqueuing a task

Equivalent execution sequences when two actions

- access variables exclusively
- resulting in equivalent task queues

Definition (Local Independence)

Given a state C, $\alpha_1, \alpha_2 \in \sum$, and $\alpha_1, \alpha_2 \in \mathit{enable}(C)$, actions α_1 and α_2 are said to be local-independent, denoted by $\alpha_1 \equiv_{LI} \alpha_2$, if the following conditions are satisfied.

1. $ex(ex(C, \alpha_1), \alpha_2) =_v ex(ex(C, \alpha_2), \alpha_1)$;
2. $Q(ex(ex(C, \alpha_1), \alpha_2)) \simeq Q(ex(ex(C, \alpha_2)), \alpha_1)$.

Local effects of an action

- updating an variable or enqueuing a task

Equivalent execution sequences when two actions

- access variables exclusively
- resulting in equivalent task queues

Independence Analysis

Local Independence

```
event void Boot.booted(){
    call Read.read();
    post send_task();
}
event void Read.rdDone(int v){
    value += v;
}
task void send_task(){
    busy = TRUE;
    call Send.send(count);
}
event void Send.sendDone(){
    busy = FALSE;
}
event void Receive.receive(){
    count++;
    post send_task();
}
```

- Let $W(t) = \{$ variables written by task $t\}$, $W(t) = \{$variables only read by task $t\}$:
- t_{rd}: $W(t_{rd}) = \{value\}$, $R(t_{rd}) = \varnothing$;
- t_{rv}: $W(t_{rv}) = \{count, busy\}$, $R(t_{rv}) = \varnothing$;
- $(W(t_{rd}) \bigcup R(t_{rd})) \bigcap W(t_{tv}) = R(t_{td}) \bigcap (W(t_{rv}) \bigcup R(t_{rv})) = \varnothing$;
- $t_{rd} \equiv_{TI} t_{rv}$ (t_{rd} is independent with task t_{rv}).

- $Q = \langle t_0, \cdots, t_n \rangle$ is a task queue;
- $Swap(Q, i) = \langle t_0, \cdots, t_{i+1}, t_i, \cdots, t_n \rangle$;

Definition (Task Queue Equivalence)

Given two task queue Q and Q', they are equivalent ($Q \simeq Q'$) iff $Q^0 = Q \wedge \exists\, m \geqslant 0, Q^m = Q' \wedge (\forall\, k \in [0, m).\ \exists\, i_k.\ t^k_{i_k} \equiv_{TI} t^k_{i_k+1} \wedge Q^{k+1} = Swap(Q^k, i_k))$ where t^k_i is the i^{th} task in Q^k.

- since $t_{rd} \equiv_{TI} t_{rv}$, thus $\langle t_{rd}, t_{rv} \rangle \simeq \langle t_{rd}, t_{rv} \rangle$.

Definition (Global Independence)

Let $t_i \in \mathit{Tasks}(\mathcal{S}_i)$ and $t_j \in \mathit{Tasks}(\mathcal{S}_j)$ such that $\mathcal{S}_i \neq \mathcal{S}_j$. Tasks t_i and t_j are said to be global-independent, denoted by $t_i \equiv_{GI} t_j$, iff $\forall \mathcal{C} \in \Gamma.\ t_i, t_j \in \mathit{EnableT}(\mathcal{C}) \Rightarrow \forall \mathcal{C}_i \in \mathit{Ex}(\mathcal{C}, t_i).\ \exists \mathcal{C}_j \in \mathit{Ex}(\mathcal{C}, t_j).\ \mathit{Ex}(\mathcal{C}_i, t_j) \asymp \mathit{Ex}(\mathcal{C}_j, t_i)$ *and vice versa*.

Informally, executing two tasks from different sensors in different orders resulting in equivalent sequences if

- there is no communication occurring in either t_i or t_j;
- if t_i sends a message to $\mathcal{S}_j$, then t_j is independent of all receive tasks of $\mathcal{S}_j$, and vice versa.

Definition (Global Independence)

Let $t_i \in Tasks(\mathcal{S}_i)$ and $t_j \in Tasks(\mathcal{S}_j)$ such that $\mathcal{S}_i \neq \mathcal{S}_j$. Tasks t_i and t_j are said to be global-independent, denoted by $t_i \equiv_{GI} t_j$, iff $\forall \mathcal{C} \in \Gamma.\ t_i, t_j \in EnableT(\mathcal{C}) \Rightarrow \forall \mathcal{C}_i \in Ex(\mathcal{C}, t_i).\ \exists \mathcal{C}_j \in Ex(\mathcal{C}, t_j).\ Ex(\mathcal{C}_i, t_j) \asymp Ex(\mathcal{C}_j, t_i)$ *and vice versa*.

Theorem (GI Detection)

$\forall t_1 \in Tasks(\mathcal{S}_i), t_2 \in Tasks(\mathcal{S}_j).\ \mathcal{S}_i \neq \mathcal{S}_j, t_1 \subset_{GI} \mathcal{S}_i \Rightarrow t_1 \equiv_{GI} t_2$, *where* $t \subset_{GI} \mathcal{S} \equiv \forall t_r \in Rcvs(\mathcal{S}), t_p \in Posts(t).\ t_r \equiv_{TI} t_p$.

- Two-level independence analysis
- State space generation by Cartesian semantics [GFYS07]
- Reduction of intra-sensor concurrency by persistent set technique [CGP01]
- Preserving LTL-X properties, i.e., LTL formulas without the X operator
- Immediately integrated with existing verification algorithms

Definition (Sensor Prefix)

A prefix $p \in \mathit{Prefix}(\mathcal{S})$ is defined as a tuple $(\langle \mathcal{C}_0, \alpha_1, \mathcal{C}_1, \cdots, \alpha_{m-1}, \mathcal{C}_m \rangle, \{br_0, br_1, \cdots, br_n\})$, where $\forall\, 1 \leq i < m.\ \alpha_i \in \sum_{\mathcal{S}} \wedge \mathcal{C}_i \stackrel{\alpha_i}{\hookrightarrow} \mathcal{C}_{i+1}$, and $\forall\, 0 \leq i \leq n.br_i \in \mathit{Prefix}(\mathcal{S}, \mathcal{C}_m)$.

Sensor Network Cartesian Semantics

Definition

Let $W(\alpha)$ be the set of variables written by an action α, and $R(\varphi)$ be the set of variables read in the target property φ.

Definition (SN Cartesian Vector)

Given a global property $\varphi \in \mathit{Gprop}$, a vector $(p_1, \cdots, p_i, \cdots, p_n)$ is a sensor network cartesian vector for $\mathcal{N}$ w.r.t. φ from a network state $\mathcal{C}$ *if the following conditions hold*:

1. $p_i \in \mathit{Prefix}(\mathcal{S}_i, \mathcal{C})$;
2. $\forall\, t \in \mathit{tasks}(p_i).\ t \not\subset_{Gl} \mathcal{S}_i \Rightarrow t \in \mathit{LastT}(p_i)$;
3. $\forall\, \alpha \in \mathit{acts}(p_i).\ \alpha \notin \mathit{safe}(\varphi) \Rightarrow \alpha \in \mathit{lastAct}(p_i)$ where $\alpha \in \mathit{safe}(\varphi)$, iff $W_\alpha \cap R(\varphi) = \varnothing$.

Exploring the state space by the SN Cartesian semantics

- handling intra-sensor concurrency
- establishing sensor prefixes
- generating sensor network Cartesian vectors (SNCVs)
- building state space by SNCVs
- applying model checking algorithms directly

Algorithms

State Exploration

- Generation of subsequent states for on-the-fly model checking algorithms:

Algorithm 1 *GetSuccessors*$(\mathcal{C}, \varphi)$

$list \leftarrow \varnothing$
$p \leftarrow pfx(\mathcal{C})$
if $Next(p, \mathcal{C}) \neq \varnothing$ **then**
    {no successors of $\mathcal{C}$ in p}
    $list \leftarrow Next(p, \mathcal{C})$
else
    {generate a new sncv from $\mathcal{C}$}
    $scv \leftarrow GetNewCV(\mathcal{C}, \varphi)$
    for all $i \leftarrow 1$ **to** n **do**
        {traverse each sensor prefix to obtain the successors of $\mathcal{C}$}
        $list \leftarrow list \cup \{Next(scv[i], \mathcal{C})\}$
    end for
end if
return *list*

Algorithms

Sensor Network Cartesian Vector Generation

Algorithm 2 $GetNewCV(\mathcal{C}, \varphi)$

$scv \leftarrow (\overbrace{\langle\rangle, \cdots, \langle\rangle}^{n})$
for all $\mathcal{S}_i \in \mathcal{N}$ **do**
  {generate prefix for sensors}
  $visited \leftarrow \{\mathcal{C}\}$
  $workingLeaf \leftarrow \varnothing$
  {generate a prefix from $\mathcal{S}$}
  $p_i \leftarrow GetPrefix(\mathcal{S}_i, \mathcal{C}, \varphi)$
  for all $lp \in leaf(p_i)$ **do**
    if $\widehat{last}(lp) \notin visited$ **and** $Extensible(lp, \mathcal{S}_i, \varphi)$ **then**
      $workingLeaf.Push(lp)$
      $visited \leftarrow visited \cup \widehat{last}(lp)$
    end if
  end for
  while $workingLeaf \neq \varnothing$ **do**
    $p_k \leftarrow workingLeaf.Pop()$
    $visited \leftarrow visited \cup \{\widehat{last}(p_k)\}$
    {generate a new prefix from the last state of p_k}
    $p'_k \leftarrow GetPrefix(\mathcal{S}_i, \widehat{last}(p_k), \varphi)$
    $p_k \leftarrow (p_k, \{p'_k\})$
    for all $lp \in leaf(p'_k)$ **do**
      if $Extensible(lp, \mathcal{S}_i, \varphi)$ **and** $\widehat{last}(lp) \notin visited$ **then**
        $workingLeaf.Push(lp)$
      end if
    end for
  end while
  {update the i^{th} element of scv with p_i}
  $scv[i] \leftarrow p_i$
end for
return scv

Algorithms

Sensor Prefix Generation

Algorithm 3 $\mathit{GetPrefix}(\mathcal{S}, \mathcal{C}, \varphi)$

$p \leftarrow \langle \mathcal{C} \rangle$
$t \leftarrow \mathit{getCurrentTsk}(\mathcal{C}, \mathcal{S})$
{extend p by executing task t}
$\mathit{ExecuteTask}(t, p, \varphi, \{\mathcal{C}\}, \mathcal{S})$
if t terminates **then**
    for all $p_i \in \mathit{leaf}(p)$ **do**
        $\mathcal{C}' \leftarrow \widehat{\mathit{last}}(p_i)$
        $\mathit{irs} \leftarrow \mathit{GetItrs}(\mathcal{C}', \mathcal{S})$
        $p_i' \leftarrow \mathit{RunItrs}(\mathcal{C}', \mathit{itrs}, \mathcal{S})$
        $p_i \leftarrow (p_i, \{p_i'\})$
    end for
end if
return p

Algorithm 4 $\mathit{ExecuteTask}(t, lp, \varphi, \mathcal{C}s, \mathcal{S})$

{let α be the current action of t}
$\alpha \leftarrow \mathit{GetAction}(t, \mathcal{C})$
$\mathcal{C} \in \widehat{last}(lp)$
{*post* actions interleave interrupts}
if $\alpha \leftarrow post(t')$ **then**
    $itrs \leftarrow \mathit{GetItrs}(\mathcal{S}, \mathcal{C})$
    {interleave α and interrupts *itrs*}
    $p \leftarrow \mathit{RunItrs}(\mathcal{C}, itrs \cup \{\alpha\}, \mathcal{S})$
    $lp \leftarrow (lp, \{p\})$
else
    {*non-post* actions}
    $\mathcal{C}' \leftarrow ex(\mathcal{C}, \alpha)$
    $tmp \leftarrow \langle \mathcal{C}, \alpha, \mathcal{C}' \rangle$
    $\mathit{setPfx}(\mathcal{C}', tmp)$
    $lp \leftarrow (lp, \{tmp\})$
end if
$lps \leftarrow leaf(lp)$
{stop executing t when t terminates or a non-safe action is encountered}
if $\alpha \notin safe(\varphi)$ **or** $terminate(t, \alpha)$ **then**
    return
end if
for all $lp' \in lps$ **do**
    {extend lp only if no loop in it}
    if $\widehat{last}(lp') \notin \mathcal{C}s$ **then**
        $\mathcal{C}s' \leftarrow \mathcal{C}s \cup states(lp')$
        {executing t to extend lp'}
        $\mathit{ExecuteTask}(t, lp', \varphi, \mathcal{C}s', \mathcal{S})$
    end if
end for

Theorem

Let $\mathcal{T}$ be the transition system of $\mathcal{N}$, where $\mathcal{N} = (\mathcal{R}, \{S_0, \cdots, S_N\})$. Let $\mathcal{T}'$ be the transition system obtained after applying the two-level partial order reduction w.r.t. φ over $\mathcal{N}$. Then $\mathcal{T}'$ and $\mathcal{T}$ are stuttering equivalent w.r.t. φ.

Theorem

Let $\mathcal{T}$ be the transition system of $\mathcal{N}$, where $\mathcal{N} = (\mathcal{R}, \{S_0, \cdots, S_N\})$. Let $\mathcal{T}'$ be the transition system obtained after applying the two-level partial order reduction w.r.t. φ over $\mathcal{N}$. Then $\mathcal{T}'$ and $\mathcal{T}$ are stuttering equivalent w.r.t. φ.

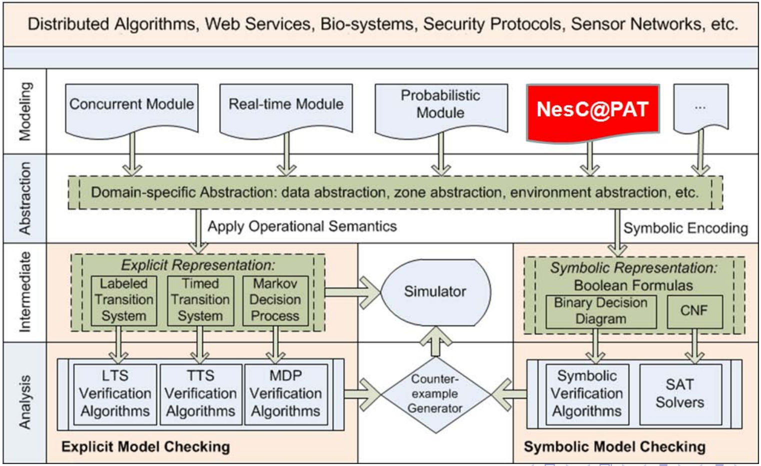
Preservation of LTL-X properties

It has been shown that if two structures $\mathcal{T}$, $\mathcal{T}'$ are stuttering equivalent w.r.t. an LTL-X property φ, then $\mathcal{T}' \models \varphi$ if and only if $\mathcal{T} \models \varphi$ [CGP01]. Therefore, our method preserves LTL-X properties.

PAT

www.patroot.com

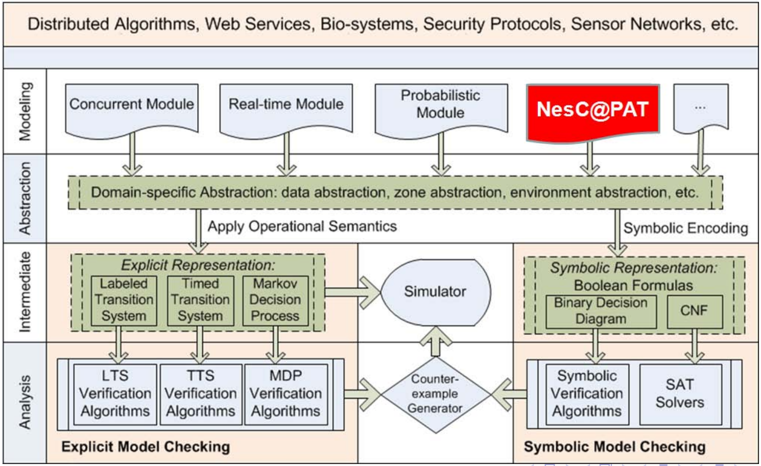
- Extensible and modularized
- Model checking algorithms for various semantic models



PAT

www.patroot.com

- More than 15 domain-specific model checkers developed on PAT
- 2300+ registered users from 550+ organizations in 58 countries



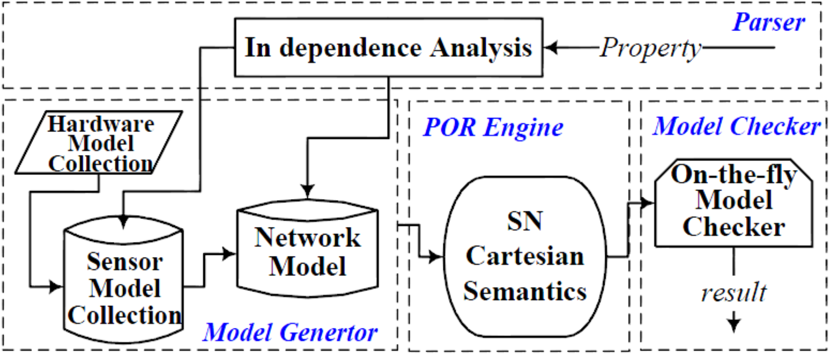
NesC@PAT

Implemented as a module in PAT model checking framework

- Fully automatic and domain-specific for NesC and WSNs
- Safety Properties
  - User-defined
  - Pre-defined low-level safety properties
    e.g, a infinite task, array index overflow, null pointer access
- Liveness (temporal) properties
  e.g, a buffer is released infinitely often

Two-level POR implementation

NesC@PAT



Motivating Example Revisited

Figure: Cartesian state space of SN with size 2

Statistics of checking deadlockfree property, i.e, $R(\varphi) = \varnothing$.

Size	**Result**	**#State**	**#Trans**	**Time(s)**	**OH(ms)**	**#States** *wo POR*	**POR Ratio**
2	✓	2040	2129	0.1	1.18	45K	0.04
3	✓	30K	31827	1.4	1.30	9M	3×10^{-3}
4	✓	276K	294K	14	2.60	2025M	1×10^{-4}
6	✓	2.3M	2.5M	129	30	9.11e+13	2.5×10^{-8}

Trickle

- LOC per sensor: 332
- Safety: false updating operation
- Livenss: $\Diamond$ *AllUpdated*

Anti-theft

- LOC per sensor: 3391
- Safety: deadlock free
- Liveness: $\Box$(theft $\Rightarrow \Diamond$alert)

App (LOC / sensor)	**Property**	**Size**	**#State**	**#Trans**	**Time(s)**	**OH**	**#States** *wo POR*	**POR Ratio**
Anti-theft (3391)	Deadlock free	3	1.2M	1.2M	791	95	>2.3G	$< 6 \times 10^{-4}$
	$\Box$(theft$\Rightarrow$ $\Diamond$alert)		1.3M	1.4M	2505	108	>4.6G	$< 3 \times 10^{-4}$
Trickle (332)	$\Diamond$ AllUpdated	2	3268	3351	3	2	111683	3×10^{-2}
		3	208K	222K	74	3	>23.7M	$< 8 \times 10^{-3}$
		4	838K	947K	405	4	>5.4G	$< 2 \times 10^{-4}$
		5	13.3M	15.7M	8591	5	>1232.2G	$< 1 \times 10^{-5}$

Table: Experiment Results of NesC@PAT with POR

- Overhead of independence analysis: negligible, within 1 second
- POR reduction ratio: at least 10^2-10^8
  - $POR\ ratio = \frac{\#State\ wt\ POR}{\#State\ wo\ POR}$
  - Safety properties: $\#State\ wo\ POR \approx S_1 \times S_2 \times \cdots \times S_n$
  - LTL properties: $\#State\ wo\ POR \approx (S_1 \times S_2 \times \cdots \times S_n) \times BA$

App (LOC / sensor)	Property	Size	#State	#Trans	Time(s)	OH	#States wo POR	POR Ratio
Anti-theft (3391)	Deadlock free	3	1.2M	1.2M	791	95	>2.3G	$< 6 \times 10^{-4}$
	$\Box$(theft$\Rightarrow$ $\Diamond$alert)		1.3M	1.4M	2505	108	>4.6G	$< 3 \times 10^{-4}$
Trickle (332)	$\Diamond$ AllUpdated	2	3268	3351	3	2	111683	3×10^{-2}
		3	208K	222K	74	3	>23.7M	$< 8 \times 10^{-3}$
		4	838K	947K	405	4	>5.4G	$< 2 \times 10^{-4}$
		5	13.3M	15.7M	8591	5	>1232.2G	$< 1 \times 10^{-5}$

Table: Experiment Results of NesC@PAT with POR

- Overhead of independence analysis: negligible, within 1 second
- POR reduction ratio: at least 10^2-10^8
  - $POR\ ratio = \frac{\#State\ wt\ POR}{\#State\ wo\ POR}$
  - Safety properties: $\#State\ wo\ POR \approx S_1 \times S_2 \times \cdots \times S_n$
  - LTL properties: $\#State\ wo\ POR \approx (S_1 \times S_2 \times \cdots \times S_n) \times BA$

App (LOC / sensor)	Property	Size	#State	#Trans	Time(s)	OH	#States wo POR	POR Ratio
Anti-theft (3391)	Deadlock free	3	1.2M	1.2M	791	95	>2.3G	$< 6 \times 10^{-4}$
	$\Box$(theft$\Rightarrow$ $\Diamond$alert)		1.3M	1.4M	2505	108	>4.6G	$< 3 \times 10^{-4}$
Trickle (332)	$\Diamond$AllUpdated	2	3268	3351	3	2	111683	3×10^{-2}
		3	208K	222K	74	3	>23.7M	$< 8 \times 10^{-3}$
		4	838K	947K	405	4	>5.4G	$< 2 \times 10^{-4}$
		5	13.3M	15.7M	8591	5	>1232.2G	$< 1 \times 10^{-5}$

Table: Experiment Results of NesC@PAT with POR

Comparison with T-Check

Checking the same safety property of Trickle

Size	NesC@PAT					T-Check					
	wt POR			*wo POR*	**Ratio**	Bound	*wt POR*			*wo POR*	**Ratio**
	#State	**Exh**	**Time(s)**				**#State**	**Exh**	**Time(s)**		
2	3012	Y	2	52.3K	6×10^{-2}	20	4765	Y	1	106.2K	$\approx 4 \times 10^{-2}$
3	120K	Y	20	$>$11.8M	$< 1 \times 10^{-2}$	12	66.2K	N	1	13.5M	$\approx 5 \times 10^{-3}$
						50	12.6M	Y	283	NA	NA
4	368K	Y	58	$>$2.7G	$< 1 \times 10^{-4}$	10	56.7K	N	1	41.8M	$\approx 1 \times 10^{-3}$
						50	420.7M	Y	1291	NA	NA
5	4.2M	Y	638	$>$616G	$< 7 \times 10^{-6}$	8	85.2K	N	1	17.4M	$\approx 1 \times 10^{-3}$
						50	NA	N	$>$12600	NA	NA

- T-Check explores more states per second
  T-Check adopts stateless model checking
- NesC@PAT requires shorter time to for state space exploration
  T-Check may explore the same path multiple times due to stateless model checking
- NesC@PAT achieves better reduction than T-Check
  T-Check only deals with network-level concurrency

Conclusion

Summary

- A two-level POR for SNs
- Preserves LTL-X properties
- Allows NesC@PAT to verify SNs with 3000+ LoC in each sensor
- Achieves good reduction results ($10^2 - 10^8$)

Future work

- Synthesis of network topology for a given property φ
- Model checking large SNs or even parameterized SNs
  - Symmetry reduction
  - Local reasoning techniques

Question

Thank you!

Bibliography I

[BK10] Doina Bucur and Marta Z. Kwiatkowska.
Software verification for TinyOS.
In *IPSN*, pages 400–401, 2010.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron Peled.
Model checking.
MIT Press, 2001.

[GFYS07] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv.
Cartesian Partial-Order Reduction.
In *SPIN*, pages 95–112, 2007.

[GLvB+03] D. Gay, P. Levis, R. v. Behren, M. Welsh, E. Brewer, and D. Culler.
The nesC Language: A Holistic Approach to Networked Embedded Systems.
In *PLDI*, pages 1–11, 2003.

Bibliography II

[LMP+04] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler.
TinyOS: An operating system for sensor networks.
In *Ambient Intelligence*. Springer Verlag, 2004.

[LR10] Peng Li and John Regehr.
T-Check: bug finding for sensor networks.
In *IPSN*, pages 174–185, Stockholm, Sweden, 2010.

[MVO+10] Luca Mottola, Thiemo Voigt, Fredrik Osterlind, Joakim Eriksson, Luciano Baresi, and Carlo Ghezzi.
Anquiro: Enabling Efficient Static Verification of Sensor Network Software.
In *SESENA*, pages 32–37, 2010.

Bibliography III

[ZSL+11] Manchun Zheng, Jun Sun, Yang Liu, Jin Song Dong, and Yu Gu.
Towards a model checker for nesc and wireless sensor networks.
In *ICFEM*, pages 372–387, 2011.