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] and Yu Gu[1]

[1] Singapore University of Technology and Design
{zmanchun,sunjun,jasongu}@sutd.edu.sg
[2] School of Computer and Statistics,Trinity College Dublin
David.Sanan@cs.tcd.ie
[3] School of Computer Engineering, Nanyang Technological University
yangliu@ntu.edu.sg
[4] School of Computing, National University of Singapore
dongjs@comp.nus.edu.sg

Abstract. Wireless sensor networks may be used to conduct critical tasks like fire detection or surveillance monitoring. It is thus important to guarantee the correctness of such systems by systematically analyzing their behaviors. Formal verification of wireless sensor networks is an extremely challenging task as the state space of sensor networks is huge, e.g., due to interleaving of sensors and intra-sensor interrupts. In this work, we develop a method to reduce the state space significantly so that state space exploration methods can be applied to a much smaller state space without missing a counterexample. Our method explores the nature of networked NesC programs and uses a novel two-level partial order reduction approach to reduce interleaving among sensors and intra-sensor interrupts. We define systematic rules for identifying dependence at sensor and network levels so that partial order reduction can be applied effectively. We have proved the soundness of the proposed reduction technique, and present experimental results to demonstrate the effectiveness of our approach.

1 Introduction

Sensor networks (SNs) are built based on small sensing devices (i.e., sensors) and deployed in outdoor or indoor environments to conduct different tasks. Recently, SNs have been applied in more areas like military surveillance, environment monitoring, theft detection, and so on [2]. Many of them are carrying out critical tasks, failures or errors of which might cause catastrophic loss of money, equipments and even human lives. Therefore, it is highly desirable that the implementation of SN systems is reliable and correct.

In order to develop reliable and correct SNs, a variety of approaches and tools have been proposed. Static analysis of SNs (e.g., [3]) is difficult, given their dynamic nature. Therefore, most of the existing approaches rely on state space exploration, e.g., through simulation [15], random walk [17], or model checking [13,19,20,23,4,5,17]. Although

* This research is partially supported by project IDG31100105/IDD11100102 from Singapore University of Technology and Design.

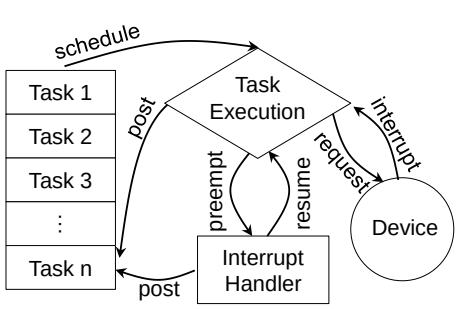
some of the tools were able to detect and reveal bugs, all of them face the same challenge: the huge state space of SNs. In practice, a typical sensor program might consist of hundreds/thousands of lines of code (LOC), which introduces a state space of tens of thousands, considering only concurrency among internal interrupts. As a result, existing tools usually cover only a fraction of the state space and/or take a long time. For instance, the work in [4,5] is limited to a single sensor, whereas the approaches in [13,19,17,25] work only for small networks. We refer the readers to Section 6 for a detailed discussion of related works.

In this work, we develop a method to significantly reduce the state space of SNs while preserving important properties so that state space exploration methods (like model checking or systematic testing) become more efficient. Our targets are SNs developed in TinyOS/NesC, since TinyOS/NesC is widely used for developing SNs. TinyOS [7] provides an interrupt-driven execution model for SNs, and NesC (Network-Embedded-System C) [10] is the programming language of TinyOS applications.

Our method is a novel two-level partial order reduction (POR) which takes advantage of the unique features of SNs as well as NesC/TinyOS. Existing POR methods [11,6,9,24,12] reduce the state space of concurrent systems by avoiding unnecessary interleaving of *independent* actions. In SNs, there are two sources of "concurrency". One is the interleaving of different sensors, which would benefit from traditional POR. The other is introduced by the internal interrupts of sensors. An interrupt can occur anytime and multiple interrupts may occur in any sequence, producing numerous states. Applying POR for interrupts is highly nontrivial because all interrupts would modify the task queue and lead to different orders of scheduled tasks at run time. Our method extends and combines two different POR methods (one for intra-sensor interrupts and one for inter-sensor interleaving) in order to achieve better reduction. We remark that applying two different POR methods in this setting is complicated, due to the interplay between inter-sensor message passing and interrupts within a sensor (e.g., a message arrival would generate interrupts).

Our method preserves both safety properties and liveness properties in the form of linear temporal logic (LTL) so that state space exploration methods can be applied to a much-smaller state space without missing a counterexample. Our method works as follows. First, static analysis is performed to automatically identify independent actions/interrupts at both inter-sensor and intra-sensor levels. Second, we extend the Cartesian semantics [12] to reduce network-level interleaving. The original Cartesian POR algorithm treats each process (in our case, sensor) as a simple sequential program. However, in our work, we have to handle the internal concurrency among interrupts for each sensor and thus the Cartesian semantics of SNs is developed. The interleaving among interrupts is then minimized by the persistent set technique [6].

We formally prove that our method is sound and complete, i.e., preserving LTL-X properties [6]. The proposed method has been implemented in the model checker NesC@PAT [25] and experiment results show that our method reduces the state space significantly, e.g., the reduced state space is orders of magnitudes smaller. We also approximated the reduction ratio obtained by a related tool T-Check [17] under POR setting and the data show that our two-level POR achieves much better reduction ratio than T-Check's POR algorithm, as elaborated in Section 5.



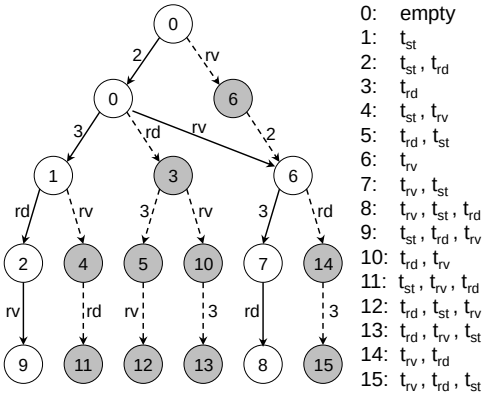
(a) TinyOS Execution Model

```
1  void handler_dev(){
2    //update status of dev;
3    ...
4    //schedule the
5    //completion event
6    post dev_compl_task();
7  }
8
9  //completion task
10 task void dev_compl_task(){
11   //event implemented
12   //by programmers
13   signal dev_done_event();
14 }
```

(b) Abstract Interrupt Handler

```
1  event void Boot.booted(){
2    call Read.read();
3    post send_task();
4  }
5  event void Read.rdDone(int v){
6    value += v;
7  }
8  task void send_task(){
9    busy = true;
10   call Send.send(count);
11 }
12 event void Send.sendDone(){
13   busy = false;
14 }
15 event void Receive.receive(){
16   count ++;
17   post send_task();
18 }
```

(c) Example Code



(d) State Graph of Event *Boot.booted*

Fig. 1. Interrupt-driven Features

2 Preliminaries

In this section, we present the interrupt-driven feature of TinyOS/NesC and the formal definitions of SNs. For details on how to generate a model from NesC programs, readers are referred to [25], which defines small step operational semantics for NesC.

In NesC programs, there are two execution contexts, *interrupt handler* and *task* (a function), described as *asynchronous* and *synchronous*, respectively [10]. An interrupt handler can always preempt a task if interrupts are enabled. In TinyOS execution model [14], a task queue schedules tasks for execution in a certain order. In our work, we model the task scheduler in the FIFO order, which is the default setting of TinyOS and is widely used. As shown in Fig. 1(a), the execution of a task could be preempted by interrupt handlers. An interrupt handler accesses low-level registers and enqueues a task to invoke a certain function at a higher level of application code. In our approach, we treat interrupt handlers as black boxes, as we assume that the low-level behaviors of devices work properly. Variables are used to represent the status of a certain device and thus low-level functions related to interrupt handlers are abstracted, as shown by

the pseudo code in Fig. 1(b). The execution of an interrupt handler is modeled as one action. Different ordering of interrupt handler executions might lead to different orders of tasks in the task queue, making the state space complex and large. In our model after a task is completed, all pending interrupt handlers are executed before a new task is loaded for execution. This approximation reduces concurrency between tasks and interrupts and is yet reasonable since devices usually respond to requests within a small amount of time like the executing period of a task.

The NesC language is an event-oriented extension of C that adds new concepts such as $call$, $signal$, and $post$. The semantics of $call$ (e.g., lines 2 and 10 in Fig. 1(c)) and that of $signal$ are similar to traditional function calls, invoking certain functions (either commands or events). The keyword $post$ (like lines 3 and 17 in Fig. 1(c)) is to enqueue a given task. Thus the task queue could be modified during both synchronous and asynchronous execution contexts. In other words, the task queue is shared by tasks and interrupt handlers. Fig. 1(c) illustrates a fragment of a NesC program, which involves messaging and sensing and is the running example of this paper. The command call $Read.read()$/$Send.send()$ invokes the corresponding command body that requests the sensing device/messaging device to read data/to send a packet, which will later trigger the completion interrupt rd/sd to post a task for signaling event $Read.rdDone$/$Send.sendDone$. We remark that rv is used to denote the interrupt of a packet arrival, and t_{rd}, t_{sd}, and t_{rv} are the tasks posted by interrupt handlers of rd, sd and rv, respectively. With the assumption that a packet arrival interrupt is possible at any time, the state graph of event $Boot.booted$ is shown in Fig. 1(d), where each transition is labeled with the line number of the executed statement or the triggered interrupt, and each state is numbered according to the task queue. The task queues of different state numbers are illustrated in Fig. 1(d) as well. For example, after executing $call\ Read.read()$ (line 2) the task queue still remains empty, while after executing the interrupt handler rv which enqueues its completion task t_{rv} and the task queue becomes $\langle t_{rv} \rangle$ (i.e., state 6).

The formal definitions of SNs are given in [25]. They are summarized below only to make the presentation self-contained.

Definition 1 (Sensor Model). *A sensor model $\mathcal{S}$ is a tuple $\mathcal{S} = (A, T, R, init, P)$ where A is a finite set of variables; T is a queue which stores posted tasks in FIFO order; R is a buffer that keeps incoming messages sent by other sensors; $init$ is the initial state of $\mathcal{S}$; and P is a program composed by the running NesC program M and interrupting devices H, i.e., $P = M \triangle H$.*

Definition 1 formally describes a sensor which runs NesC programs. Let $\mathcal{S}$ be a sensor. A state C of $\mathcal{S}$ is a tuple (V, Q, B, P) where V is the current valuation of variables declared by the NesC programs of $\mathcal{S}$; Q is a sequence of tasks scheduled in the task queue; B is the sequence of packets in the message buffer; and P is the running program. In this work, we use $V(C)$, $Q(C)$, $B(C)$ and $P(C)$ to denote the variable valuation, task queue, message buffer and running program of a state C, respectively.

A sensor transition t is defined as $C \xrightarrow{\alpha}_s C'$, where C (C') is the state before and after executing the action α, represented as $C' = ex(C, \alpha)$. We define $enable(C)$ to be the set of all actions enabled at state C, i.e., $enable(C) = \{\alpha \mid \exists\, C' \in \mathbb{C}, C \xrightarrow{\alpha} C'\}$. Further, $ex(C, \alpha)$ (where $\alpha \in enable(C)$) denotes the state after executing α at state C.

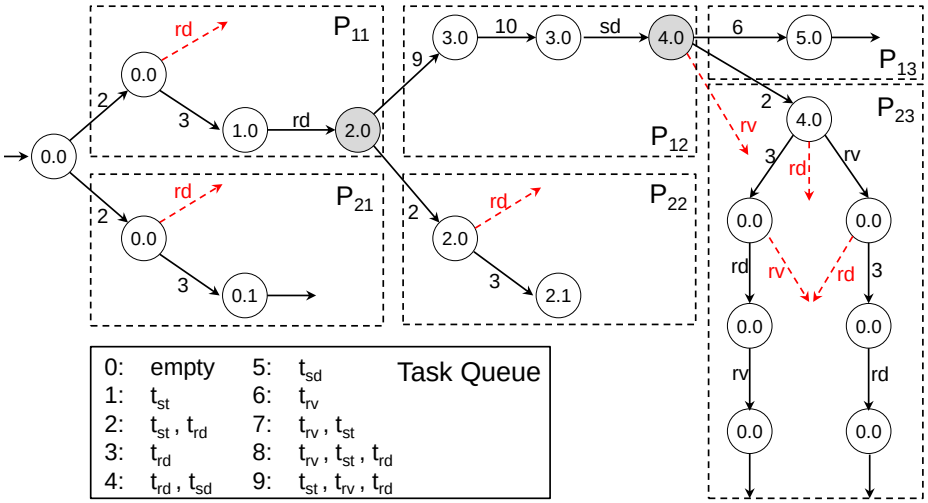


Fig. 2. Pruned State Graph

$\sum_{\mathcal{S}}$ (or simply $\sum$ if $\mathcal{S}$ is clear) denotes the set of actions of $\mathcal{S}$. We define $itrQ(\mathcal{S}) \subseteq \sum$ as the set of hardware request actions and $sd(\mathcal{S})$ as the set of actions involving packet transmission. $Tasks(\mathcal{S})$ (or simply $Tasks$ if $\mathcal{S}$ is clear) denotes the set of all tasks defined in $\mathcal{S}$. For a given NesC program, we assume that $\sum$ and $Tasks$ are finite.

Definition 2 (Sensor Network Model). *A sensor network model $\mathcal{N}$ is defined as a tuple $(\mathcal{R}, \{\mathcal{S}_0, \cdots, \mathcal{S}_n\})$ where $\mathcal{R}$ is the network topology, and $\{\mathcal{S}_0, \cdots, \mathcal{S}_n\}$ is a finite ordered set of sensor models, with $\mathcal{S}_i$ $(0 \leqslant i \leqslant n)$ being the i^{th} sensor model.*

A state $\mathcal{C}$ of a sensor network is defined as an ordered set of states $\{C_1, \cdots, C_n\}$ where C_i $(1 \leqslant i \leqslant n)$ is the state of $\mathcal{S}_i$, denoted by $\mathcal{C}[i]$. A sensor network transition $\mathcal{T}$ is defined as $\mathcal{C} \overset{\alpha}{\hookrightarrow} \mathcal{C}'$ where $\mathcal{C}$ ($\mathcal{C}'$) is the state before (after) the transition, represented as $\mathcal{C}' = Ex(\mathcal{C}, \alpha)$. In the following of this paper, a state of a sensor is referred to as a *local* state, whereas a state of a sensor network is called a *global* state or simply a state.

3 Two-level Independence Analysis

Inside a sensor, the interleaving between an interrupt handler and a non-post action can be reduced, since interrupt handlers only modify the task queue and non-post actions never access the task queue. For example, in Fig. 1(d), the interleaving between line 2 and rv can be ignored. Moreover, for post statements and interrupt handlers, their interleaving could be reduced if their corresponding tasks access no common variables. For example, t_{rd} only accesses variable $value$ which is never accessed by t_{rv}, so the interleaving between interrupt handlers rd and rv at state 1 can be alleviated. In Fig. 1(d), dashed arrows and shadow states stand for transitions and states that can be pruned.

Therefore, it is important to detect the independence among actions inside a sensor, referred to as $local\ independence$.

Among a sensor network, each sensor only accesses its own and local resources, unless it sends a message packet, modifying some other sensors' message buffers. Intuitively, the interleaving of local actions of different sensors can be reduced without missing critical states. This observation leads to the independence analysis at the network level, referred to as $global\ independence$.

Consider a network with two sensors $\mathcal{S}_1$ and $\mathcal{S}_2$ implemented with the code shown in Fig. 1(c). Applying partial order reduction at both network and sensor levels, we are able to obtain a reduced state graph as shown in Fig. 2. States are numbered with the task queues of both sensors. For example, state 2.1 shows that the task queue of $\mathcal{S}_1$ is $\langle t_{st}, t_{rd} \rangle$ and $\langle t_{st} \rangle$ for $\mathcal{S}_2$. In this example, interleaving between the two sensors is only allowed when necessary, like at the shadow states labeled with 2.0 and 4.0. The sub-graph within each dashed rectangle is established by executing actions from only one sensor, either $\mathcal{S}_1$ or $\mathcal{S}_2$. In each sub-graph, local independence is applied to avoid unnecessary interleaving among local actions. Dashed arrows indicate pruned local actions. For example, rectangle P_{23} is constructed by removing all shadow states and dashed transitions in Fig. 1(d). The corresponding complete state space of this graph consists of around 200 states, whereas the reduced graph contains fewer than 20 states.

The shape of the reduced state graph might be different according to the property being checked. For example, if the property is affected by the value of the variables $busy$ and $value$ of the example in Fig. 1(d), then the interleaving between rd and rv can not be avoided. Therefore, we need to investigate local and global independence w.r.t. a certain property, and in the following of this paper, the concepts of independence, equivalence and so on are discussed w.r.t. a certain property φ. We present the definitions of local independence and global independence in Section 3.1 and 3.2, respectively, both with rules for identifying them.

3.1 Local Independence

In a sensor, an action may modify a variable or the task queue. Local independence is defined by the effects on the variables and the task queue. In the following, the concepts of actions, tasks, and task queues are w.r.t. a given sensor $\mathcal{S}$.

Definition 3 (Local Independence). *Given a local 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. $Q(ex(ex(C, \alpha_1), \alpha_2)) \simeq Q(ex(ex(C, \alpha_2)), \alpha_1)$.

In the above definition, $=_v$ denotes that two local states share the same valuation of variables, message buffer, and the same running program. That is, if $C_1 = (V_1, Q_1, B_1, P_1)$, $C_2 = (V_2, Q_2, B_2, P_2)$, then we have $C_1 =_v C_2$ iff $V_1 = V_2 \wedge B_1 = B_2 \wedge P_1 = P_2$. If only the first condition in Definition 3 is satisfied, α_1 and α_2 are said to be *variable-independent*, denoted by $\alpha_1 \equiv_{VI} \alpha_2$. The relation $\simeq$ will be covered in Definition 6. Let W_α and R_α be the set of variables written and only read by an action α, respectively.

Lemma 1. $\forall\, \alpha_1, \alpha_2 \in \sum .\ W_{\alpha_1} \cap (W_{\alpha_2} \cup R_{\alpha_2}) = W_{\alpha_2} \cap (W_{\alpha_1} \cup R_{\alpha_1}) = \varnothing \Rightarrow \alpha_1 \equiv_{VI} \alpha_2$. □

Lemma 1 [1] shows that two actions are variable-independent if the variables modified by one action are mutually exclusive with those accessed (either modified or read) by the other. For example, $\alpha_{l6} \equiv_{VI} \alpha_{l13}$, where $\alpha_{l6}(\alpha_{l13})$ refers to the action executing the statement at line 6 (13) of Fig. 1(c). Furthermore, we can conclude that a non-post action in the synchronous context is always local-independent with any action in the asynchronous context [1]. This is shown in Lemma 2.

Lemma 2. $\forall\, \alpha \in \sum^{syn}, \alpha' \in \sum^{asyn} . \alpha \notin \sum^{pt} \Rightarrow \alpha \equiv_{LI} \alpha'$. □

Inside a sensor, interrupt handlers might run in parallel, which produces different orders of tasks in the task queue. Given a task t, $Ptask(t)$ denotes the set of tasks posted by a post statement in t or an interrupt handler of a certain interrupt request in t. Formally, $Ptask(t) = \{t' \mid \exists\, \alpha \in t.\ \alpha = post(t') \vee (\alpha \in itrQ(\mathcal{S}) \wedge t' = tsk(ih(\alpha)))\}$, where $post(t)$ is a post statement to enqueue task t; $ih(\alpha_{iq})$ denotes the corresponding interrupt handler of a device request α_{iq}, and $tsk(\alpha_{ih})$ denotes the completion task of α_{ih}. In the code in Fig. 1(c), $Ptask(t_{rv}) = \{t_{st}\}$, due to the *post* statement in line 17. As for t_{st} (lines 8 to 11), it has a request for sending a message (line 10), the interrupt handler of which will post the task t_{sd}, and thus $Ptask(t_{st}) = \{t_{sd}\}$.

Since more tasks can be enqueued during the execution of a previously enqueued task, we define $Rtask(t)$ to represent all possible tasks enqueued by a given task t and the tasks in its $Ptask$ set in a recursive way. Formally, $Rtask(t) = \{t\} \cup Ptask(t) \cup (\cup_{t' \in Ptask(t)} Rtask(t'))$. Since $Tasks$ is finite, for every task t, $Rtask(t)$ is also finite and thus could be obtained statically at compile time. In Fig. 1(c), since $Ptask(t_{sd}) = \varnothing$, we have $Rtask(t_{sd}) = \{t_{sd}\}$. Similarly, we can obtain that $Rtask(t_{st}) = \{t_{st}, t_{sd}\}$ and $Rtask(t_{rv}) = \{t_{rv}, t_{st}, t_{sd}\}$. Let $R(\varphi, \mathcal{S})$ be the set of variables of $\mathcal{S}$ accessed by the property φ. Let $\widehat{W}(t)$ be the set of variables modified by any task in $Rtask(t)$. We say that t is a φ-safe task, denoted by $t \in safe(\varphi, \mathcal{S})$ iff $(\widehat{W}(t) \cap R(\varphi, \mathcal{S})) = \varnothing$.

Definition 4 (Local Task Independence). *Let $t_{i(j)} \in Tasks$ be two tasks. t_i and t_j are said to be local-independent, denoted by $t_i \equiv_{TI} t_j$, iff $(t_i \in safe(\varphi, \mathcal{S}) \vee t_j \in safe(\varphi, \mathcal{S})) \wedge \forall\, t_i' \in Rtask(t_i), t_j' \in Rtask(t_j).\ \forall\, \alpha_i \in t_i', \alpha_j \in t_j'.\ \alpha_i \equiv_{VI} \alpha_j$.*

Though interrupt handlers and post statements might modify the task queue concurrently, we observe that task queues with different orders of tasks might be equivalent. Based on Definition 4, we define the independence relation of two task sequences, which is used to further define equivalent task sequences.

Definition 5 (Task Sequence Independence). *Let $Q_i = \langle t_{i0}, \cdots, t_{im}\rangle$, $Q_j = \langle t_{j0}, \cdots, t_{jn}\rangle (m, n \geqslant 0)$ be two task sequences, where $t_{iu} (0 \leq u \leq m), t_{jv} (0 \leq v \leq n) \in Tasks$. Q_i and Q_j are said to be sequence-independent, denoted by $Q_i \equiv_{SI} Q_j$, iff $\forall\, t_i \in (\cup_{k=0}^{m} Rtask(t_{ik})), t_j \in (\cup_{k=0}^{n} Rtask(t_{jk}))$. $t_i \equiv_{TI} t_j$.*

Let $q_1 \frown q_2$ be the concatenation of two sequences q_1 and q_2. A partition $\mathcal{P}$ of a task sequence Q is a list of task sequences $q_0, q_1, \cdots, q_m$ such that $Q = q_0 \frown q_1 \frown \cdots q_m$,

and for all $0 \leqslant i \leqslant m$, $q_i \neq \langle\rangle$ (q_i is called a sub-sequence of Q). We use $part(Q)$ to denote the set of all possible partitions of Q. Given a partition $\mathcal{P}$ of Q such that $Q = q_0 \frown q_1 \frown \cdots \frown q_n$, $Swap(Q, i) = q_0 \frown \cdots q_{i+1} \frown q_i \frown \cdots \frown q_n$ denotes the task sequence obtained by swapping two adjacent sub-sequences (i.e., q_i and q_{i+1}) of Q.

Definition 6 (Task Sequence Equivalence). *Two task sequences Q and Q' are equivalent ($Q \simeq Q'$) iff $Q^0 = Q \ \wedge \ \exists\, m \geqslant 0.\ Q^m = Q' \ \wedge \ (\forall\, 0 \leq k < m.\ (\exists\, i_k.\ Q^{k+1} = Swap(Q^k, i_k) \ \wedge \ q^k_{i_k} \equiv_{SI} q^k_{i_k+1}))$ where q^k_i is the i^{th} sub-sequence of Q^k.*

The above definition indicates that if a task sequence Q' can be obtained by swapping adjacent independent sub-sequences of Q, then $Q \simeq Q'$. Given two local states C and C', we said that C is equivalent to C', denoted by $C \cong C'$, iff $C =_v C' \wedge Q(C) \simeq Q(C')$. Further, two local state sets $\mathbb{C}$, $\mathbb{C}'$ are said to be equivalent, denoted by $\mathbb{C} \asymp \mathbb{C}'$, iff $\forall\, C \in \mathbb{C}.\ \exists\, C' \in \mathbb{C}'.\ C \cong C'$ and vice versa. We explore the execution of task sequences starting at a local state which is the completion point of a previous task, i.e., a local state with the program as $(\checkmark \ \triangle \ H)$ [25]. This is because that only after a task terminates can a new task be loaded from the task queue for execution. The case when $B(C) \neq \langle\rangle$ is related to network communication, and is ignored here but will be covered in global independence analysis in Section 3.2.

Lemma 3. *Given $C = (V, Q, \langle\rangle, \checkmark \ \triangle \ H)$ and $C' = (V, Q', \langle\rangle, \checkmark \ \triangle \ H)$, let $exs(Q_i, C_i)$ be the set of final local states after executing all tasks of Q_i starting at local state C_i. $Q \simeq Q' \Rightarrow exs(Q, C) \asymp exs(Q', C')$.* □

Lemma 3 shows that executing two equivalent task sequences from v-$equal$ local states will always lead to equivalent sets of final local states, as proved in [1]. Given an action α, we use $ptsk(\alpha)$ to denote the set of tasks that could be enqueued by executing α. With the above lemma, the rule for deciding local independence between actions can be obtained by Lemma 4 [1].

Lemma 4. *Given C, $\alpha_1, \alpha_2 \in enable(C)$, $(\alpha_1 \equiv_{VI} \alpha_2 \ \wedge \forall\, t_1 \in ptsk(\alpha_1), t_2 \in ptsk(\alpha_2).\ t_1 \equiv_{TI} t_2) \Rightarrow \alpha_1 \equiv_{LI} \alpha_2$.* □

3.2 Global Independence

SNs are non-blocking, i.e., the execution of one sensor never blocks others. In addition, a sensor accesses local resources most of the time, except when it broadcasts a message to the network and fills in others' message buffers. At the network level, we explore the execution of each sensor individually, and only allow interleaving among sensors when an action involving network communication is performed. Let $\mathcal{N}$ be a sensor network with n sensors $\mathcal{S}_1, \mathcal{S}_2 \cdots \mathcal{S}_n$ and $\mathcal{C}$ be a global state. We use $EnableT(\mathcal{C})$ to denote the set of enabled tasks at $\mathcal{C}$. Given $t \in Tasks(\mathcal{S}_i)$, $t \in EnableT(\mathcal{C}) \Leftrightarrow \mathcal{C}[i] = (V, \langle t, \cdots\rangle, B, \checkmark \ \triangle \ H)$. $Ex(\mathcal{C}, t)$ represents the set of final states after executing task t (and interrupt handlers caused by it) starting from $\mathcal{C}$. For two global states $\mathcal{C}_1$ and $\mathcal{C}_2$, we say that $\mathcal{C}_1$ and $\mathcal{C}_2$ are equivalent ($\mathcal{C}_1 \cong \mathcal{C}_2$) iff $\forall\, 1 \leq i \leq n.\ \mathcal{C}_1[i] \cong \mathcal{C}_2[i]$. Similarly, we say that two sets of global states Γ and Γ' are equivalent (i.e., $\Gamma \asymp \Gamma'$) iff $\forall\, \mathcal{C} \in \Gamma.\ \exists\, \mathcal{C}' \in \Gamma'.\ \mathcal{C} \cong \mathcal{C}'$ and vice versa.

Definition 7 (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.*

A data transmission would trigger a packet arrival interrupt at the receivers and thus is possible to interact with local concurrency inside sensors. In the following, $Sends(\mathcal{S})$ denotes the set of tasks that contain data transmission requests, and $Rcvs(\mathcal{S})$ denotes the set of completion tasks of packet arrival interrupts.

Given $t \in Tasks(\mathcal{S})$, t is considered as rcv-independent, denoted by $t \subset_{RI} \mathcal{S}$, iff $\forall\, t_r \in Rcvs(\mathcal{S}), t_p \in Posts(t).\ t_r \equiv_{TI} t_p$. A rcv-independent task never posts a task local-dependent with the completion task of any packet arrival interrupts. Thus, we can ignore interleaving such tasks with other sensors even if there exists data transmission. We say that t is a *global-safe* task of $\mathcal{S}$, i.e., $t \subset_{GI} \mathcal{S}$, iff $t \subset_{RI} \mathcal{S}$. If $t \not\subset_{GI} \mathcal{S}$, then t is *global-unsafe*. The following theorem indicates that a global-safe task is always global-independent with any task of other sensors [1].

Theorem 1. $\forall\, t_1 \in Tasks(\mathcal{S}_i), t_2 \in Tasks(\mathcal{S}_j).\ \mathcal{S}_i \neq \mathcal{S}_j, t_1 \subset_{GI} \mathcal{S}_i \Rightarrow t_1 \equiv_{GI} t_2.$ □

4 SN Cartesian Partial Order Reduction

In this section, we present our two-level POR, which extends the Cartesian vector approach [12] and combines it with a persistent set algorithm [11].

4.1 Sensor Network Cartesian Semantics

Cartesian POR was proposed by Gueta et. al. to reduce non-determinism in concurrent systems, which delays unnecessary context switches among processes [12]. Given a concurrent system with n processes and a state s, a Cartesian vector is composed by n prefixes, where the i^{th} $(1 \leq i \leq n)$ prefix refers to a trace executing actions only from the i^{th} process starting from state s. For SNs, sensors could be considered as concurrent processes and their message buffers could be considered as "global variables".

It has been shown that Cartesian semantics is sound for local safety properties [12]. A global property that involves local variables of multiple processes (or sensors) is converted into a local property by introducing a dummy process for observing involved variables. In our case, we avoid this construction by considering global property in the Cartesian semantics for SNs. Let $Gprop(\mathcal{N})$, or simply $Gprop$ since $\mathcal{N}$ is clear in this section, be the set of *global* properties defined for $\mathcal{N}$. Given an action $\alpha \in Tasks(\mathcal{S})$ and a global property $\varphi \in Gprop$, α is said to be *φ-safe*, denoted by $\alpha \in safe(\varphi)$, iff $W_\alpha \cap R(\varphi) = \varnothing$ where W_α is the set of variables modified by α and $R(\varphi)$ is the set of variables accessed by φ. If $\alpha \notin safe(\varphi)$, then α is said to be *φ-unsafe*.

In order to allow sensor-level nondeterminism inside prefixes, we redefine $Prefix$ as a "trace tree" rather than a sequential trace. Let $Prefix(\mathcal{S})$ be the set of all prefixes of sensor $\mathcal{S}$, $Prefix(\mathcal{S}, \mathcal{C})$ be the set of prefixes of $\mathcal{S}$ starting at $\mathcal{C}$, and $first(p)$ be the initial state of a prefix p. A prefix is defined as follows.

Definition 8 (Prefix). *A prefix $p \in Prefix(\mathcal{S})$ is defined as a tuple $(trunk, branch)$, where $trunk = \langle \mathcal{C}_0, \alpha_1, \mathcal{C}_1, \cdots, \alpha_{m-1}, \mathcal{C}_m \rangle \wedge m \geq 0 \wedge \forall 1 \leq i < m.\ \alpha_i \in \sum_{\mathcal{S}} \wedge \mathcal{C}_i \xrightarrow{\alpha_i} \mathcal{C}_{i+1}$, and $branch \subseteq Prefix(\mathcal{S}, \mathcal{C}_m)$, being a set of prefixes of $\mathcal{S}$.*

Let $p \in Prefix(\mathcal{S})$ and $p = (p_{tr}, \{p_{b1}, p_{b2}, \cdots, p_{bm}\})$. We define $tr(p)$ to denote the trunk of prefix p before branching prefixes (i.e., $tr(p) = p_{tr}$), and $br(p)$ to denote the set of branching prefixes of p (i.e., $br(p) = \{p_{b1}, p_{b2}, \cdots, p_{bm}\}$). In Fig. 2, the dashed rectangles p_{11}, p_{12} and p_{13} are prefixes of $\mathcal{S}_1$, and p_{21}, p_{22} and p_{23} are prefixes of $\mathcal{S}_2$. More specifically, $tr(p_{23}) = \langle (4.0), \alpha_2, (4.0) \rangle$ and $br(p_{23}) = \{\langle (4.0), \alpha_3, (4.1), \alpha_{rd}, \cdots \rangle, \langle (4.0), \alpha_{rv}, (4.6), \alpha_3, \cdots \rangle\}$. Given a prefix $p \in Prefix(\mathcal{S})$, the following notations are defined:

- The set of states in p: $states(p) = \{\mathcal{C}_0, \cdots, \mathcal{C}_m\} \cup (\cup_{sp \in br(p)} states(sp))$.
- The set of *leaf* prefixes of $\mathcal{S}$: $LeafPrefix(\mathcal{S}) = \{lp \mid \forall\, lp \in Prefix(\mathcal{S}).\ br(lp) = \varnothing\}$. Given $lp \in LeafPrefix(\mathcal{S})$, $\widehat{last}(lp)$ denote the last state of lp.
- The set of *tree* prefixes of $\mathcal{S}$: $TreePrefix(\mathcal{S}) = Prefix(\mathcal{S}) - LeafPrefix(\mathcal{S})$.
- The set of leaf prefixes of p: $p \in LeafPrefix(\mathcal{S}) \Rightarrow leaf(p) = p \wedge p \in TreePrefix(\mathcal{S}) \Rightarrow leaf(p) = \cup_{bp \in br(p)} leaf(bp)$.
- The set of final states of p: $p \in LeafPrefix(\mathcal{S}) \Rightarrow last(p) = \{\widehat{last}(p)\} \wedge p \in TreePrefix(\mathcal{S}) \Rightarrow last(p) = \cup_{bp \in br(p)} last(bp)$.
- Subsequent prefixes $\sqsupset$: $\forall\, lp \in LeafPrefix(\mathcal{S}).\ lp \sqsupset p \equiv \widehat{last}(lp) = first(p)$.
- Concatenation of leaf prefixes $\widehat{\ }$: $\forall\, p1 = \langle \mathcal{C}_0, \alpha_0, \cdots, \mathcal{C}_k \rangle, p2 = \langle \mathcal{C}_k, \alpha_k, \mathcal{C}_{k_0}, \alpha_{k_0}, \cdots, \mathcal{C}_{k_m} \rangle . p1 \widehat{\ } p2 = \langle \mathcal{C}_0, \alpha_0, \cdots, \mathcal{C}_k, \alpha_k, \mathcal{C}_{k_0}, \cdots, \mathcal{C}_{k_m} \rangle$.

We also define $tasks(p)$ ($acts(p)$) to denote the set of tasks (actions) executed in p. Moreover, $lastT(p)$ ($lastAct(p)$) denotes the set of last tasks (actions) executed in p.

Definition 9 (SN Cartesian Vector). *Given a global property $\varphi \in Gprop$, a vector $(p_1, \cdots, p_i, \cdots, p_n) \in Prefix^n$ is a sensor network Cartesian vector for $\mathcal{N}$ w.r.t. φ from a state $\mathcal{C}$* if the following conditions hold*:*

1. $p_i \in Prefix(\mathcal{S}_i, \mathcal{C})$;
2. $\forall\, t \in tasks(p_i).\ t \not\subset_{GI} \mathcal{S}_i \Rightarrow t \in LastT(p_i)$;
3. $\forall\, \alpha \in acts(p_i).\ \alpha \notin safe(\varphi) \Rightarrow \alpha \in lastAct(p_i)$.

According to Definition 9, a vector $(p_0, p_1, \cdots, p_n)$ from $\mathcal{C}$ is a valid sensor network Cartesian vector (SNCV) if for every $0 \leq i \leq n$, p_i is a prefix of $\mathcal{S}_i$ and each leaf prefix of p_i ends with a φ-unsafe action or a global-unsafe task as defined in Section 3.2. Furthermore, we define the corresponding inference rules of SNCVs [1]. In Fig. 2, if $value, busy \notin R(\varphi)$, then (p_{11}, p_{21}) is a valid SNCV from the initial state.

4.2 Two-level POR Algorithm

In this section, we present the two-level POR algorithm. The main idea is to explore the state space by the sensor network Cartesian semantics and to perform reduction during the generation of SNCVs. First, we present the top-level state exploration algorithm, which could be invoked in existing verification algorithms directly without changing the verification engine. Second, we show the algorithms for SNCV generation, as well as algorithms for producing a sensor prefix.

Algorithm 1 State Space Generation

$GetSuccessors(\mathcal{C}, p, \varphi)$

1: $list \leftarrow \varnothing$
2: **if** $Next(p, \mathcal{C}) \neq \varnothing$ **then**
3: $\quad list \leftarrow Next(p, \mathcal{C})$
4: **else**
5: $\quad scv \leftarrow GetNewCV(\mathcal{C}, \varphi)$
6: $\quad$ **for all** $i \leftarrow 1$ **to** n **do**
7: $\quad\quad list \leftarrow list \cup \{Next(scv[i], \mathcal{C})\}$
8: $\quad$ **end for**
9: **end if**
10: **return** $list$

- **State Space Generation**. Given a state $\mathcal{C}$, a prefix p ($\mathcal{C} \in states(p)$) and a global property φ, the state space of $\mathcal{N}$ is explored via a corresponding SNCV, as shown in Algorithm 1. In this algorithm, $GetNewCV(\mathcal{C}, \varphi)$ generates a new SNCV from $\mathcal{C}$, which will be explained later. The relation $Next : Prefix(\mathcal{S}) \times \Gamma \rightarrow \mathbb{P}(\Gamma)$ traverses a prefix to find a set of successors of $\mathcal{C}$. Formally, $Next(p, \mathcal{C}) = \{\mathcal{C}' \mid \exists\, \alpha \in acts(p), \mathcal{C} \overset{\alpha}{\hookrightarrow} \mathcal{C}'\}$. The function $ConcatTree$ extends a leaf prefix with another prefix as its branch, defined as $ConcatTree(lp \in LeafPrefix(\mathcal{S}), sp \in Prefix(\mathcal{S}))$. Formally, if $lp \sqsupset sp$, after executing $ConcatTree(lp, sp)$, we have $lp' = (lp, \{sp\})$. We remark that $ConcatTree(lp, sp)$ has a side effect in lp by updating it with the resultant prefix of the combination.
- **SNCV Generation**. Algorithm 2 is dedicated to SNCV generation, i.e., the method $GetNewCV$. In this algorithm, $visited$ is the set of final states of prefixes that have been generated, and $workingLeaf$ is the stack of leaf prefixes to be further extended. Concurrency at network level is minimized by lines 7 and 18, where the relation $Extensible : Prefix(\mathcal{S}) \times \{\mathcal{S}_1, \cdots, \mathcal{S}_n\} \times Gprop \rightarrow \{True, False\}$ is defined as $Extensible(p, \mathcal{S}, \varphi) \equiv \forall\, t \in lastT(p), \alpha \in lastAct(p).\ t \subset_{GI} \mathcal{S} \wedge \alpha \in safe(\varphi) \wedge \alpha \notin sd(\mathcal{S})$. In other words, a prefix is further extended (lines 15 to 21) only if it has not executed a global-unsafe task, a φ-unsafe action or a messaging action. The function $GetPrefix(\mathcal{S}_i, \mathcal{C}, \varphi)$ produces a prefix of $\mathcal{S}_i$ by executing actions and interrupt handlers of $\mathcal{S}_i$ in parallel. At first, p_i is initialized by $GetPrefix(\mathcal{S}_i, \mathcal{C}, \varphi)$, and is then extended by recursively concatenating each of its leaf prefixes with a new prefix obtained by $GetPrefix(\mathcal{S}_i, \mathcal{C}', \varphi)$, as shown by lines 12 to 22. If p_i is inextensible, it is assigned to the i^{th} element of the sensor Cartesian vector scv ($scv[i]$) by line 23.
- **Sensor Prefix Generation**. Algorithm 3 shows how a sensor $\mathcal{S}$ establishes a prefix from $\mathcal{C}$ w.r.t. φ. Function $ExecuteTask(t, p, \varphi, \mathcal{C}s, \mathcal{S})$ extends the initial prefix p by executing actions in task t, until a φ-unsafe action or a loop is encountered. Interrupt handlers are delayed as long as the action being executed is a non-post statement, which is reasonable due to Lemma 1 and Lemma 4. A persistent set approach has been adopted in both $ExecuteTask$ and $RunItrs$ to constrain interleaving to happen only between local-dependent actions.
- **Task Execution.** In Algorithm 4, the following notations are used.
  - Set $\mathcal{C}s$: the set of states that has been visited.
  - Method $GetAction(t, \mathcal{C})$: returns the enabled action of task t at state $\mathcal{C}$.
  - Method $setPfx(\mathcal{C}, p)$: assigns prefix p as the prefix that state $\mathcal{C}$ belongs to.

  Initially, the currently enabled action α will be executed (lines 5 to 16). At this phase, two cases are considered. The first is when α is a $post$ statement, and inter-

Algorithm 2 Sensor Network Cartesian Vector Generation

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

1: $scv \leftarrow (\overbrace{\langle\rangle, \cdots, \langle\rangle}^{n})$
2: **for all** $\mathcal{S}_i \in \mathcal{N}$ **do**
3: $visited \leftarrow \{\mathcal{C}\}$
4: $workingLeaf \leftarrow \varnothing$
5: $p_i \leftarrow GetPrefix(\mathcal{S}_i, \mathcal{C}, \varphi)$
6: **for all** $lp \in leaf(p_i)$ **do**
7: **if** $Extensible(lp, \mathcal{S}_i, \varphi)$ **and** $\widehat{last}(lp) \notin visited$ **then**
8: $workingLeaf.Push(lp)$
9: $visited = visited \cup \widehat{last}(lp)$
10: **end if**
11: **end for**
12: **while** $workingLeaf \neq \varnothing$ **do**
13: $p_k \leftarrow workingLeaf.Pop()$
14: $visited \leftarrow visited \cup \{\widehat{last}(p_k)\}$
15: $p'_k \leftarrow GetPrefix(\mathcal{S}_i, \widehat{last}(p_k), \varphi)$
16: $ConcatTree(p_k, p'_k)$
17: **for all** $lp \in leaf(p'_k)$ **do**
18: **if** $Extensible(lp, \mathcal{S}_i, \varphi)$ **and** $\widehat{last}(lp) \notin visited$ **then**
19: $workingLeaf.Push(lp)$
20: **end if**
21: **end for**
22: **end while**
23: $scv[i] \leftarrow p_i$
24: **end for**
25: **return** scv

Algorithm 3 Prefix Generation

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

1: $p \leftarrow \langle \mathcal{C} \rangle$
2: $t \leftarrow getCurrentTsk(\mathcal{C}, \mathcal{S})$
3: $ExecuteTask(t, p, \varphi, \{\mathcal{C}\}, \mathcal{S})$
4: **if** t is finished **then**
5: **for all** $p_i \in leaf(p)$ **do**
6: $\mathcal{C}' \leftarrow \widehat{last}(p_i)$
7: $irs \leftarrow GetItrs(\mathcal{C}', \mathcal{S})$
8: $p'_i \leftarrow RunItrs(\mathcal{C}', itrs)$
9: $ConcatTree(p_i, p'_i)$
10: **end for**
11: **end if**
12: **return** p

rupts dependent with α will be taken to run in parallel in order to preserve states with different task queues. This is achieved by lines 5 to 9. The second case handles all non-$post$ actions, and the action will be executed immediately to obtain the resultant prefix (lines 10 to 16). In this case, all interleaving between interrupts and the action α is ignored, which is reasonable by Lemma 2. After the action α completes its execution, the algorithm will return immediately if α is φ-unsafe or t has no more actions to be executed. Otherwise, a new iteration of $ExecuteTask$ will be invoked at each final state of the prefix that has been currently established (lines 22 to 29). Line 24 is to prevent the algorithm to be stuck by loops.

- **Interleaving Interrupts.** Algorithms 5 and 6 show how partial order reduction could be applied at sensor level to alleviate interleaving caused by concurrency among tasks and interrupts. The idea is motivated by the observation that the only shared resource among interrupt handlers and normal actions is the task queue. By Lemma 2, there are two kinds of concurrency to be considered, i.e. the concurrency between a $post$ statement, and the concurrency between any two interrupt handlers. Algorithm 5 ($RunItrs$) establishes a prefix for the sensor $\mathcal{S}$ from a state $\mathcal{C}$ by interleaving actions in the set $itrs$ using a persistent set approach. Here, $itrs$ would be a set of interrupt handlers plus at most one $post$ action. Algorithm 6 (Persistent Set) establishes a persistent set from a given set of actions $itrs$. If $itrs$ contains a

Algorithm 4 Task Execution

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

1: {let α be the current action of t}
2: $\alpha \leftarrow GetAction(t, \mathcal{C})$
3: $\mathcal{C} \in \widehat{last}(lp)$
4: {only *post* actions need to interleave interrupts}
5: **if** $\alpha \leftarrow post(t')$ **then**
6: $itrs \leftarrow GetItrs(\mathcal{S}, \mathcal{C})$
7: {interleave α and interrupts $itrs$}
8: $p \leftarrow RunItrs(\mathcal{C}, itrs \cup \{\alpha\}, \mathcal{S})$
9: $lp \leftarrow (lp, \{p\})$
10: **else**
11: {*non-post* actions run independently}
12: $\mathcal{C}' \leftarrow ex(\mathcal{C}, \alpha)$
13: $tmp \leftarrow \langle \mathcal{C}, \alpha, \mathcal{C}' \rangle$
14: $setPfx(\mathcal{C}', tmp)$
15: $lp \leftarrow (lp, \{tmp\})$
16: **end if**
17: $lps \leftarrow leaf(lp)$
18: {stop executing t when t terminates or a non-safe action is encountered}
19: **if** $\alpha \notin safe(\varphi)$ **or** $terminate(t, \alpha)$ **then**
20: **return**
21: **end if**
22: **for all** $lp' \in lps$ **do**
23: {extend lp only if there is no loop in it}
24: **if** $\widehat{last}(lp') \notin \mathcal{C}s$ **then**
25: $\mathcal{C}s' \leftarrow \mathcal{C}s \cup states(lp')$
26: {continue to execute t to extend lp'}
27: $ExecuteTask(t, lp', \varphi, \mathcal{C}s', \mathcal{S})$
28: **end if**
29: **end for**

Algorithm 5 Interleaving Interrupts

$RunItrs(\mathcal{C}, itrs, \mathcal{S})$

1: **if** $itrs \leftarrow \varnothing$ **then**
2: **return** $\langle\rangle$
3: **end if**
4: $p \leftarrow \langle \mathcal{C} \rangle$
5: {pis is the persistent set of $itrs$}
6: $pis \leftarrow GetPerSet(itrs, \mathcal{C}, \mathcal{S})$
7: {interleave dependent actions}
8: **for all** $\alpha \in pis$ **do**
9: $\mathcal{C}' \leftarrow ex(\mathcal{C}, \alpha)$
10: $lp \leftarrow \langle \mathcal{C}, \alpha, \mathcal{C}' \rangle$
11: $setPfx(\mathcal{C}', lp)$
12: {only allow interleaving if α is not a *post*}
13: **if** $\alpha \in \sum^{iq}$ **then**
14: $s \leftarrow RunItrs(\mathcal{C}', pis - \{\alpha\}, \mathcal{S})$
15: $lp \leftarrow (lp, \{s\})$
16: **end if**
17: {add lp as a new branch to p}
18: $p \leftarrow (tr(p), br(p) \cup \{lp\})$
19: **end for**
20: **return** p

post action, then this *post* action will be chosen as the first action of the persistent set to return; otherwise, an action will be chosen randomly to start generating the persistent set (lines 2 to 10). After that, the persistent set will be extended by iteratively adding actions from $itrs$ that are dependent with at least one action in the persistent set.

4.3 Correctness

In the following, we show that the above POR algorithms work properly and are sound for model checking global properties and LTL-X properties. Lemmas 5 and 6 assure the correctness of the functions invoked in Algorithm 3, which are proved in [1].

Lemma 5. *Given a state $\mathcal{C}$ where $\mathcal{C}[i] = (V, Q, B, \checkmark \triangle H)$, $RunItrs(\mathcal{C}, Get\text{-}Itrs(\mathcal{C}', \mathcal{S}_i))$ terminates and returns a valid prefix of $\mathcal{S}_i$.* □

Algorithm 6 Persistent Set

$GetPerSet(itrs, \mathcal{C}, \mathcal{S})$

1: {choose an α to start with}
2: **if** $\exists\, \alpha' \in itrs.\ \alpha \notin \sum^{iq}$ **then**
3: {there exists a $post$ in $itrs$, then we should start from the $post$}
4: $\alpha \leftarrow \alpha'$
5: **else**
6: **if** $\alpha' \in itrs$ **then**
7: {choose an α form $itrs$ randomly}
8: $\alpha \leftarrow \alpha'$
9: **end if**
10: **end if**
11: $pset \leftarrow \{\alpha\}$
12: $work \leftarrow \{\alpha\}$
13: **while** $work \neq \varnothing$ **do**
14: $\alpha \leftarrow work.Pop()$
15: {find new dependent actions of α from $itrs$}
16: $\alpha s \leftarrow DepActions(\alpha, itrs - pset)$
17: $pset \leftarrow pset \cup \alpha s$
18: $work \leftarrow work \cup \alpha s$
19: **end while**
20: **return** $pset$

Lemma 6. *Given $t \in Tasks(\mathcal{S})$, $t \in EnableT(\mathcal{C})$ and φ, $ExecuteTask(t, \langle\mathcal{C}\rangle, \varphi, \{\mathcal{C}\})$ extends $\langle\mathcal{C}\rangle$ by executing actions in t and enabled interrupt handlers, until t terminates or a φ-unsafe action or a loop is encountered.* □

Based on Lemma 5 and 6, we can show the correctness of Algorithm 2 in generating a prefix for a given state and a property, as shown in the following theorem.

Theorem 2. *Given $\mathcal{S}$, $\mathcal{C}$ and φ, Algorithm 3 terminates and returns a valid prefix of $\mathcal{S}$ for some SNCV.*

Proof By Lemma 6, after line 3 p is a valid prefix of $\mathcal{S}$. If lines 5 to 10 are not executed, then p is immediately returned. Suppose lines 5 to 10 are executed, and at the beginning of the i^{th} iteration of the "for" loop p is a valid prefix. Let $\widehat{p}$ be the updated prefix after line 9, and then $leaf(\widehat{p}) = (leaf(p) - p_i) \cup leaf(p_i')$ since p_i has been concatenated with p_i'. By Lemma 5, p_i' is a valid prefix and thus $\widehat{p}$ is a valid prefix. Therefore, at the beginning of the $(i+1)^{th}$ iteration, p is a valid prefix. By Lemmas 6 and 5, both lines 3 and 8 terminate. Further, we assume that variables are finite-domain, and thus the size of $leaf(p)$ is finite assuring that the "for" loop terminates. □

Theorem 3. *For every state $\mathcal{C}$, Algorithm 2 terminates and returns a valid sensor network Cartesian vector.*

Proof We prove that at the beginning of each iteration of the "while" loop (lines 12 to 22) in Algorithm 2 the following conditions hold for any i ($1 \leq i \leq n$):

1. $p_i \in Prefix(\mathcal{S}_i, \mathcal{C})$;
2. $workingLeaf = \{p \in leaf(p_i) \mid Extensible(p, \mathcal{S}_i, \varphi) \wedge \widehat{last}(p) \notin visited\}$.

By line 5, it is immediately true that $first(p_i) = \mathcal{C}$. Since $ConcatTree$ never changes the first state of a given prefix, $first(p_i) = \mathcal{C}$ holds for all iterations. Since p_i is extended by $GetPrefix(\mathcal{S}_i, \widehat{last}(p_k), \varphi)$ (line 15), which only executes actions of $\mathcal{S}_i$, thus $p_i \in Prefix(\mathcal{S}_i)$ always holds. Intuitively, condition 1 holds for all iterations. Condition 2 can be proved by induction, as the following.

At the first iteration, by lines 6 to 11, we can immediately obtain that $workingLeaf = \{lp \in leaf(p_i) \mid Extensible(lp, \mathcal{S}_i, \varphi) \wedge \widehat{last}(lp) \notin visited\}$ and condition 2 holds.

Suppose that at the beginning of the m^{th} iteration, condition 2 holds with $workingLeaf = w_m$, $p_i = p_m$. After executing line 13, we can obtain that $workingLeaf = w_m - \{p_k\}$. By lines 15 to 21, $wokingPrefix = (w_m - \{p_k\}) \cup \{lp \in leaf(p'_k) \mid Extensible(lp, \mathcal{S}_i, \varphi) \wedge \widehat{last}(lp) \notin visited\}$ (1). Let $\widehat{p}_k$ be the new value of p_k after executing line 16, by the definition of $ConcatTree$, we have $\widehat{p}_k = (p_k, p'_k)$ and thus $leaf(\widehat{p}_k) = leaf(p'_k)$ (2). Consequently, we have $leaf(p_i) = (leaf(p_m) - \{p_k\}) \cup leaf(\widehat{p}_k)$, since the leaf prefix p_k has been extended to be a tree prefix $\widehat{p}_k$. Since $w_m = \{p \in leaf(p_m) \mid Extensible(p, \mathcal{S}_i, \varphi) \wedge \widehat{last}(p) \notin visited\}$, with (1) and (2), we can obtain that at the beginning of the $(m+1)^{th}$ iteration condition 2 holds. By the definition of $Extensible$, we can conclude that $\forall\, t \in tasks(p_i), \alpha \in acts(p_i).\ t \not\subset_{GI} \mathcal{S}_i \Rightarrow t \in LastT(p_i) \wedge (\alpha \notin safe(\varphi) \vee \alpha \in sd(\mathcal{S}_i)) \Rightarrow \alpha \in lastAct(p_i)$ holds when the while loop terminates. Thus the Cartesian vector generated by Algorithm 2 is valid.

Further, by the definition of $Extensible$ we can conclude that $\forall\, t \in tasks(p_i), \alpha \in acts(p_i).\ t \not\subset_{GI} \mathcal{S}_i \Rightarrow t \in LastT(p_i) \wedge (\alpha \notin safe(\varphi) \vee \alpha \in \sum^{sd}) \Rightarrow \alpha \in lastAct(p_i)$ holds when the while loop terminates. Thus the Cartesian vector generated by Algorithm 2 is valid. As for termination, we assume that all variables are finite-domain and thus the state space is finite. On one hand, the function $GetPrefix(\mathcal{S}, \mathcal{C}, \varphi)$ always terminates and returns a valid prefix, which has been proved in Theorem 2. On the other hand, Algorithm 2 uses $visited$ to store states that have been used to generate new prefixes, and by lines 7 and 18 a state is used at most once to generate a new prefix, and termination guaranteed. □

Let φ be a property and ψ be the set of propositions belonging to φ. In the following, we discuss the stuttering equivalent relation of different objects w.r.t. ψ and the notation is simplified as stuttering equivalent when ψ is clear.

Let $L(\mathcal{C})$ be the valuation of the truth values of ψ in state $\mathcal{C}$. Given two traces $\sigma = \mathcal{C}_0, \alpha_0, \mathcal{C}_1, \alpha_1, \cdots, \mathcal{C}_i, \alpha_i, \cdots$ and $\sigma' = \mathcal{C}'_0, \alpha'_0, \mathcal{C}'_1, \alpha'_1, \cdots, \mathcal{C}'_i, \alpha'_i, \cdots$, they are referred to as stuttering equivalent, i.e., $\sigma \equiv_{st_\psi} \sigma'$, iff $L(\mathcal{C}_0) = L(\mathcal{C}'_0)$ and for every integer set $M = \{m_0, m_1, \cdots, m_i\}$ $(i \geq 0)$, there exists another integer set $P = \{p_0, p_1, \cdots, p_i\}$, such that $m_0 = p_0 = 0 \wedge$ for all $0 \leq k < i$, there exists $n_k, q_k > 0$ such that $m_{k+1} = m_k + n_k \wedge L(\mathcal{C}_{m_k}) = L(\mathcal{C}_{m_k+1}) = \cdots = L(\mathcal{C}_{m_k+(n_k-1)}) \neq L(\mathcal{C}_{m_{k+1}}) \wedge p_{k+1} = p_k + q_k \wedge L(\mathcal{C}_{m_k}) = L(\mathcal{C}'_{p_k}) = L(\mathcal{C}'_{p_k+1}) = \cdots = L(\mathcal{C}'_{p_k+(q_k-1)}) \wedge L(\mathcal{C}'_{p_k+1}) = L(\mathcal{C}_{m_{k+1}})$, and vice versa.

Let $exc(\mathcal{C}, \mathcal{S})$ be the set of traces obtained by executing only actions from $\mathcal{S}$ following the original semantics. Given a prefix p, we define $traces(p)$ to be the set of traces that could be obtained by p. Formally, $traces(p) = p \in LeafPrefix \wedge traces(p) = \{tr(p)\} \vee \{\sigma \mid \exists\, bp \in br(p).\ \sigma' \in traces(bp) \Rightarrow \sigma = tr(p) + \sigma'\}$, where "+" concatenates two traces. Lemma 7 illustrates that Algorithm 3 returns a prefix of traces stuttering equivalent to those generated by the original semantics. It shows that for all possible local interleaving from $\mathcal{C}$ for a certain sensor $\mathcal{S}$, the sensor prefix obtained by $GetPrefix$ contains the same sequences of valuations for the set of propositions ψ of the property φ. We refer interested readers to [1] for the proof of this lemma.

Lemma 7. *Given a state $\mathcal{C}$, let $p = GetPrefix(\mathcal{S}, \mathcal{C}, \varphi)$ be the prefix obtained by Algorithm 3. For all $\sigma \in exc(\mathcal{C}, \mathcal{S})$, there exists $\sigma' \in traces(p)$ such that $\sigma \equiv_{st_\psi} \sigma'$, and vice versa.* □

Two transition systems $\mathcal{T}$ and $\mathcal{T}'$ are said to be stuttering equivalent w.r.t. φ iff $\mathcal{C}_0 = \mathcal{C}'_0$ where $\mathcal{C}_0(\mathcal{C}'_0)$ is the initial state of $\mathcal{T}(\mathcal{T}')$, and for every trace σ in $\mathcal{T}$ there exists a trace σ' in $\mathcal{T}'$ such that $\sigma \equiv_{st_\psi} \sigma'$, and vice versa. In the following, we prove that the transition system obtained by the two-level POR approach is stuttering equivalent with the transition system obtained by the original sensor network semantics.

Theorem 4. *Given a sensor network $\mathcal{N} = (\mathcal{R}, \{\mathcal{S}_0, \cdots, \mathcal{S}_n\})$, let $\mathcal{T}$ be the transition system of $\mathcal{N}$ by the original semantics and 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.*

Proof Let ψ be the set of propositions contained in φ, and let $\mathcal{C}_0$, $\mathcal{C}'_0$ be the initial state of $\mathcal{T}$, $\mathcal{T}'$, respectively. It is immediately true that $\mathcal{C}_0 = \mathcal{C}'_0$ because both $\mathcal{T}$ and $\mathcal{T}'$ are obtained from the initial state of $\mathcal{N}$. We will prove that for any trace $\sigma = \mathcal{C}_0, \alpha_0, \cdots, \alpha_m, \mathcal{C}_{m+1}$ from $\mathcal{T}$, there exists a trace $\sigma' = \mathcal{C}'_0, \alpha'_0, \cdots, \alpha'_m, \mathcal{C}'_{m+1}$ in $\mathcal{T}'$ such that $\sigma \equiv_{st_\psi} \sigma'$. This will be proved by induction in the number of updating the valuation of ψ in a certain trace σ.

Base Case: if the number of updating the valuation of ψ in σ is zero, then we have $L(\mathcal{C}_0) = \cdots = L(\mathcal{C}_{m+1})$. Since $\mathcal{C}_0$ belongs to $\mathcal{T}'$, then let $\sigma' = \mathcal{C}'_0$ and $\sigma' \equiv_{st_\psi} \sigma$.

Induction Step: suppose that when the number of updating the valuation of ψ in σ is x, there exits $\sigma' = \mathcal{C}_0, \alpha'_0 \cdots \alpha'_n, \mathcal{C}_{m+1}$ such that $\sigma \equiv_{st_\psi} \sigma'$, and it also holds for the case when there are $(x+1)$ changes in the valuation of ψ in σ.

Let $\alpha_{k_1}, \alpha_{k_2}, \cdots, \alpha_{k_x}$ be the actions in σ such that $\alpha_{k_i} \notin \mathit{safe}(\varphi)$ for all $1 \leq i \leq x$. Suppose that there exist $l_1, \cdots, l_x$ such that for all $1 \leq i \leq x$, $0 \leq l_i \leq n \ \wedge \ \alpha_{k_i} \in \sum_{\mathcal{S}_{l_i}}$. Suppose that α_{k_i} is the last extendable action from a task $t_{k_i} \in \mathit{Tasks}(S_{l_i})$ such that $t_{k_i} \not\subset_{GI} S_{l_i}$ (1) where each k_i is ordered as follows. Given two last extendable actions $\alpha_{k_a} \in \sum_{\mathcal{S}_{l_a}}$, $\alpha_{k_b} \in \sum_{\mathcal{S}_{l_b}}$ $(l_a \neq l_b)$, if α_{k_a} is a Send action and $\alpha_{k_b} \in t$, $t \not\subset_{GI} \mathcal{S}_{l_b}$ then there exists α'_{k_b} which is a receive interrupt handler in S_{l_a}. If $a < b' < b$ then $k_b < k_a$, otherwise $k_a > k_b$. The same reasoning is applied when α_{k_b} is Send action and $\alpha_{k_a} \in t$, $t \not\subset_{GI} \mathcal{S}_{\alpha_{k_a}}$. If α_{k_a} and α_{k_b} are both Send actions or they belong to a task $t \not\subset_{GI} \mathcal{S}_{l_b}$ then $k_a > k_b$ if $a > b$ and vice verse.

By independence of global actions two consecutive actions $\alpha_{s-1} \in \Sigma_{S_l}, \alpha_s \notin \Sigma_{S_l}$ can be permuted for all $0 \leq s \leq k_0$ and the trace $\langle \cdots, \mathcal{C}_{s-1}, \alpha_{s-1}, \mathcal{C}_s, \alpha_s, \mathcal{C}_{s+1}, \cdots \rangle$ is equivalent to the trace $\langle \cdots, \mathcal{C}'_{s-1}, \alpha_s, \mathcal{C}'_s, \alpha_{s+1}, \mathcal{C}'_{s+1}, \cdots \rangle$. It is possible to get a trace $\sigma_{k_0} = \mathcal{C}_0, \alpha'_0, \cdots, \alpha_{k_0}, \mathcal{C}'_{k_0}$ such that for $0 \leq j \leq k_0$, $\alpha_j \in \Sigma_{S_l}$. Let $cv = (p_0, \cdots p_l, \cdots, p_n) = \mathit{GetNewCV}(\mathcal{C}_0)$. By Algorithm 2, Theorem 3 and Lemma 7, there exits a trace $\sigma'_{k_0} \in \mathit{traces}(p_l)$ such that $\sigma'_{k_0} = \mathcal{C}_0, \alpha''_0, \cdots, \mathcal{C}''_{k_0}$ and $\sigma_{k_0} \equiv_{st_\varphi} \sigma'$. Repeating this for all k_i in (1) and by transitivity of stuttering [18] we get that $\sigma_{k_x} = \mathcal{C}_0, \cdots, \alpha_{k_0}, \cdots, \alpha_{k_x}, \mathcal{C}_{k_x+1} \equiv_{st_\varphi} \sigma'_{k_x} = \mathcal{C}_0, \cdots, \alpha'_{k_0}, \cdots, \alpha'_{k_x}, \mathcal{C}_{k_x+1}$. Permuting again $\cdots, \mathcal{C}_{s-1}, \alpha_{s-1}, \mathcal{C}_s, \alpha_s, \mathcal{C}_{s+1}, \cdots$, for all $k_x \leq s \leq k_{x+1}$ and by Algorithm 2, Theorem 3, and Lemma 7 $\sigma_i = \mathcal{C}_0, \cdots, \alpha_{k_1}, \cdots, \alpha_{k_{x+1}}, \mathcal{C}_{k_{x+1}+1} \equiv_{st_\varphi} \sigma'_i = \mathcal{C}_0, \cdots, \alpha'_{k_1}, \cdots, \alpha'_{k_{x+1}}, \mathcal{C}'_{k_{x+1}+1}$ and the number of changes in the valuations of ψ is $(x+1)$. By I.H. and transitivity of stuttering $\sigma \equiv_{st_\psi} \sigma'$. □

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 $\mathcal{T} \models \varphi$ [6]. Therefore, our method preserves LTL-X properties.

App (LOC / sensor)	Property	Size	#State	#Trans	Time(s)	OH(ms)	#States *wo POR*	POR Ratio
Anti-theft (3391)	Deadlock free	3	1.2M	1.2M	791	95	>2.3G	$< 6 \times 10^{-4}$
	□(theft⇒ ◇alert)		1.3M	1.4M	2505	108	>4.6G	$< 3 \times 10^{-4}$
Trickle (332)	◇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 1. Experiment Results with NesC@PAT

5 Experiments and Discussion

We implemented our approach in NesC@PAT [25], a domain-specific model checker for sensor networks implemented using NesC programs. Static analysis is conducted at compile time to identify the global and local independence relations among actions and tasks, and then Algorithm 1 is adopted for state space exploration. In this section, we first evaluate the performance of the two-level POR method using a number of real-world SN applications. Then a comparison between our POR and the POR implemented in T-Check [17] is provided, since T-Check provides verification of TinyOS/NesC programs with a POR algorithm. All necessary materials for re-running the experiments can be obtained from [1].

5.1 Enhancing NesC@PAT with Two-level POR

First, we used NesC@PAT to model check an anti-theft application and the Trickle algorithm [16]. The anti-theft application is taken from the TinyOS distribution, in which each sensor runs a NesC program of over 3000 LOC. The Trickle algorithm is widely used for code propagation in SNs, and we adopted a simplified implementation to show the reduction effects. For the anti-theft application, we checked if a sensor turns on its theft led whenever a theft is detected, i.e., □(theft ⇒ ◇alert). Deadlock checking was also performed for anti-theft. As for the Trickle algorithm, we checked that eventually all the nodes are updated with the latest data among the network, i.e., $\Diamond AllUpdated$.

Verification results are presented in Table 1. Column OH shows the computational overhead for static analysis, which is dependent on LOC, network size and the property to be checked. This overhead is negligible (within 1 second) even for a large application like Anti-theft. The second last column estimates the complete state space size and we calculate the reduction ratio as $POR\ ratio$ (=$\frac{\#State\ wt\ POR}{\#State\ wo\ POR}$). For safety properties, $\#State\ wo\ POR$ is estimated as $S_1 \times S_2 \cdots \times S_n$, where S_i is the state space of the i^{th} sensor; as for LTL properties, it is further multiplied by the size of the Büchi automaton of the corresponding LTL property. Note that this estimation of $\#State\ wo\ POR$ is an under approximation since the state space of a single sensor is calculated without networked communication. Therefore, the $POR\ Ratio$ (both in Table 1 and 2) is also an under approximation. Therefore, our POR approach achieves a reduction of at least 10^2-10^6. Further, the larger a network is, the more reduction it will achieve.

#Node	NesC@PAT					T-Check					
	wt POR			#State *wo POR*	Ratio	#Bound	*wt POR*			#State *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

Table 2. Comparison with T-Check

5.2 Comparison with T-Check

In this section, we compared the performance of our POR approach and that of T-Check, by checking the same safety property for the Trickle algorithm, on the same testbed with Ubuntu 10.04 instead of Windows XP. The safety property is to guarantee that each node never performs a wrong update operation. We focused on reachability analysis as T-Check lacks support of LTL. We approximated the POR ratio obtained by T-Check by the number of states explored, i.e., $POR\ Ratio \approx \frac{\#State\ wt\ POR}{\#State\ wo\ POR}$, because T-Check adopts stateless model checking. Moreover, there is no way to calculate the complete state space of a single sensor and thus it is difficult to estimate the complete state space like what we did for NesC@PAT. Thus, we had to set small bounded numbers (around 10) in order to obtain the number of states explored by T-Check without the POR setting. The results indicate that for small networks with two or three nodes, both approaches gain similar POR ratio, but for larger networks with over four nodes, our approach outperforms T-Check significantly. We present the comparison of both approaches in Table 2, where Exh indicates if all states are explored. The POR method of T-Check treats all actions within the same sensor as $dependent$, i.e., it only reduces inter-sensor concurrency. Thus, our two-level approach would be able to obtain better reduction since intra-sensor concurrency is also minimized. Another observation is that T-Check explores more states per second, which is reasonable since T-Check does not maintain all explored states. However, our approach is more efficient in state space exploration, taking shorter time (10^2-10^3). This is mainly because T-Check may explore the same path multiple times due to its stateless model checking.

6 Related Work

This work is related to tools/methods on exploring state space of SNs.

Approaches like SLEDE [13] and the work by McInnes [19] translate NesC programs into formal description techniques (FDT) like $Promela$ (supported by SPIN) or CSP_M (supported by FDR) and use existing model checkers to conduct verification tasks. Anquiro [20] translates Conitiki C code into $Bogor$ models and uses BOGOR to perform the verification. Anquiro [20] is built based on the Bogor model checking framework [21,22], for model checking WSN software written in C language for Contiki OS [8]. Source codes are firstly abstracted and converted to Anquiro-specific models, i.e., Bogor models with domain-specific extensions. Then Bogor is used to model

check the models against user-specified properties. Anquiro provides three levels of abstraction to generate Anquiro-specific models and state hashing technique is adopted to reduce state space, and thus Anquiro is able to verify a network with hundreds of nodes within half an hour. However, since many low-level behaviors are abstracted away, Anquiro might not be able to detect certain bugs. Moreover, translation-based approaches could cause inaccurate results due to the semantic difference between NesC and FDTs. Hence, approaches for direct verifying NesC programs have been developed.

Werner et. al. studied the $ESAWN$ protocol by producing abstract behavior models from TinyOS applications, and used CBMC to verify the models [23]. The original $ESAWN$ consists of 21000 LOC, and the abstract behavior model contains 4400 LOC. Our approach is comparable to this approach, since we support SNs with thousands of LOC per sensor. Werner's work is dedicated to checking the $ESAWN$ protocol and it abstracts away all platform-related behaviors. Tos2CProver [4,5] translates embedded C code to standard C to be verified by CBMC, and a POR approach is integrated. Our work differs from this work in that Tos2CProver only checks single-node TinyOS applications instead of the whole network. T-Check [17] is built on TOSSIM [15] and checks the execution of SNs by DFS or random walk to find a violation of safety properties. T-Check adopts stateless and bounded model checking and is efficient to find bugs, and it helped to reveal several unknown bugs. However, T-Check might consume a large amount of time (days or weeks) to find a violation if a large bounded number is required due to the (equivalently) complete state space exploration. T-Check applies POR at network level to reduce the state space and our approach complements it with a more effective POR which preserves LTL-X.

This work is also related to research on partial order reduction in general. Approaches that using static analysis to compute a sufficient subset of enabled actions for exploration are proposed, such as persistent/sleep set [11] and ample set [6] approaches. There are also dynamic methods which compute persistent sets of transitions on the fly [9,24]. A Cartesian POR [12] was presented to delay context switches between processes for concurrent programs.

7 Conclusions

In conclusion, we proposed a two-level POR to reduce the state space of SNs significantly, based on the independence of actions. We extended the Cartesian semantics to deal with concurrent systems with multiple levels of non-determinism such as SNs. POR was then achieved by static analysis of independence and the sensor network Cartesian semantics. We also showed that it preserves LTL-X properties. We implemented this two-level POR approach in the NesC model checker NesC@PAT and it had significantly improved the performance of verification, by allowing sensor networks with thousands of LOC in each sensor to be model checked exhaustedly, with a reduction ratio sometimes more than 10^6. One of our future directions is to apply abstraction techniques like [20] to obtain an abstracted model before applying POR to support large networks with hundreds of nodes, and another is to adopt BDD techniques to implement symbolic model checking.

References

1. Experiment Materials. http://www.comp.nus.edu.sg/~pat/NesC/por .
2. I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: a Survey. *Computer networks*, 38(4):393–422, 2002.
3. W. Archer, P. Levis, and J. Regehr. Interface contracts for TinyOS. In *IPSN*, pages 158–165, Massachusetts, USA, 2007.
4. D. Bucur and M. Z. Kwiatkowska. Bug-Free Sensors: The Automatic Verification of Context-Aware TinyOS Applications. In *AMI*, pages 101–105, Salzburg, Austria, 2009.
5. D. Bucur and M. Z. Kwiatkowska. On software verification for sensor nodes. *Journal of Systems and Software*, 84(10):1693–1707, 2011.
6. E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 2001.
7. D. E. Culler, J. Hill, P. Buonadonna, R. Szewczyk, and A. Woo. A Network-Centric Approach to Embedded Software for Tiny Devices. In *EMSOFT*, pages 114–130, 2001.
8. A. Dunkels, B. Grönvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *LCN*, pages 455–462, 2004.
9. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121. ACM, 2005.
10. 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.
11. P. Godefroid and P. Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. *Formal Methods in System Design*, 2(2):149–164, 1993.
12. G. Gueta, C. Flanagan, E. Yahav, and M. Sagiv. Cartesian Partial-Order Reduction. In *SPIN*, pages 95–112, 2007.
13. Y. Hanna, H. Rajan, and W. Zhang. SLEDE: a domain-specific verification framework for sensor network security protocol implementations. In *WISEC*, pages 109–118, 2008.
14. P. Levis and D. Gay. *TinyOS Programming*. Cambridge University Press, 1 edition, 2009.
15. P. Levis, N. Lee, M. Welsh, and D. E. Culler. TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications. In *SenSys*, pages 126–137, 2003.
16. P. Levis, N. Patel, D. E. Culler, and S. Shenker. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *NSDI*, pages 15–28, California, USA, 2004.
17. P. Li and J. Regehr. T-Check: bug finding for sensor networks. In *IPSN*, pages 174–185, Stockholm, Sweden, 2010.
18. B. Luttik and N. Trcka. Stuttering congruence for *chi*. In *SPIN*, pages 185–199, 2005.
19. A. I. McInnes. Using CSP to Model and Analyze TinyOS Applications. In *ECBS*, pages 79–88, California, USA, 2009.
20. L. Mottola, T. Voigt, F. Osterlind, J. Eriksson, L. Baresi, and C. Ghezzi. Anquiro: Enabling Efficient Static Verification of Sensor Network Software. In *SESENA*, pages 32–37, 2010.
21. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC / SIGSOFT FSE*, pages 267–276, 2003.
22. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: A Flexible Framework for Creating Software Model Checkers. In *TAIC PART*, pages 3–22, 2006.
23. F. Werner and D. Faragó. Correctness of Sensor Network Applications by Software Bounded Model Checking. In *FMICS*, pages 115–131, 2010.
24. Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. In *SPIN*, volume 5156 of *LNCS*, pages 288–305. Springer, 2008.
25. M. Zheng, J. Sun, Y. Liu, J. S. Dong, and Y. Gu. Towards a model checker for nesc and wireless sensor networks. In *ICFEM*, pages 372–387, 2011.