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

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

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

Abstract. 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 like model checking or systematic testing) 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. Applying partial order reduction for sensor network programs is highly non-trivial, due to the interplay between inter-sensor message passing and interrupts or among 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

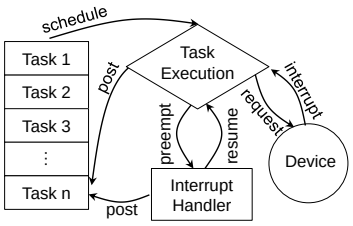
Sensor networks (SNs) are built based on small sensing devices (i.e., sensors) and deployed in outdoor or indoor environments to conduct different tasks. Recently, SNs have been experiencing an increasing application for various purposes, e.g., military surveillance, environment monitoring, theft detection, and so on [2]. Such systems are usually expected to run without human surveillance for a long period like months or even years. Many of them are carrying out critical tasks, failures or errors of which might cause a catastrophic loss in money, time, or even

human life. Therefore, it is highly desired that sensor network implementations are reliable and correct.

In order to develop reliable and correct sensor networks, a variety of approaches and tools have been proposed. Static analysis of sensor networks (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 some of the tools were able to detect and reveal bugs, all of them face the same challenge: the huge state space of sensor networks. Given a network consisting of n sensors, each of which has m states, the size of the state space is in the order of $n^m \cdot 2^{C(n,2)}$ where $2^{C(n,2)}$ is the number of network topologies. In practice, a typical sensor program might consist of hundreds/thousands of lines of code (LOC) and has a state space of tens of thousands, considering only internal interrupts. As a result, existing tools often 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. The solutions in [20,23] rely on aggressive abstraction techniques and thus is limited to particular properties only. Furthermore, T-Check [17] which is based on stateless model checking takes days or even months to detect a faulty state. We refer the readers to Section 7 for the detailed discussion of the related works.

In this work, we develop a method to significantly reduce the state space of sensor networks while preserving important properties so that state space exploration methods (like model checking or systematic testing) become more effective. Our targets are sensor networks developed in TinyOS/NesC, since TinyOS and NesC are widely used in developing SNs. The operating system TinyOS [7] provides an interrupt-driven execution model for SNs, with a library of hardware services like data transmission, timer, etc. The programming language NesC (Network-Embedded-System C) [10] is used to develop TinyOS applications, with a component-based programming pattern.

Our method is a novel two-level partial order reduction (POR) which takes advantage of unique features of sensor networks 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 sensor networks, there are two sources of "concurrency". One is the interleaving of different sensors, which would benefit from traditional POR. The other is the intra-sensor interrupts. An interrupt can occur anytime and multiple interrupts may occur in any sequence. As a result, interrupts generate a large number of states, just as interleaving. To apply POR for interrupts is highly nontrivial because all interrupts would modify the task queue and lead to different orders of scheduled tasks at runtime. Our method extends and combines two different POR methods (one for intra-sensor interrupts and one for inter-sensor interleaving) in order to achieve maximum 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 retrieval would generate interrupts).



(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

Fig. 1. Interrupt-driven Features

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. Firstly, static analysis is performed to automatically identify independent actions at both inter-sensor and intra-sensor levels, based on the rules presented in Section **??**. We propose a systematic way to detect independence among interrupts by taking into account the shared task queue. Secondly, we extend the cartesian semantics [12] in order 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 handle intra-sensor concurrency caused by interrupts and thus the cartesian semantics of SNs is different from the original one. The persistent set technique [6] is used to minimize interleaving caused by interrupts inside each sensor. The two-level POR algorithm is presented in Section **??**.

We formally prove that our method is sound and complete, i.e., preserving LTL-X properties which are LTL formulas without the X operator. The proposed method has been implemented in the model checker NesC@PAT [25], which analyzes properties of sensor networks by exploring the state space of NesC programs. We have evaluated the efficiency of our method with a number of SNs. The results show that our method reduces the size of the state space significantly, e.g., the reduced state space is thousands of times smaller or even more depending on the size of networks. We also approximated the reduction ratio gained by T-Check [17] under POR setting and the data show that our two-level POR obtains much better reduction ratio than T-Check's POR algorithm, as elaborated in Section 6.

2 Preliminaries

In this section, we firstly present the interrupt-driven feature of TinyOS/NesC and then the formal definitions of sensor networks. For details on how to generate

a model from actual NesC programs, readers can refer to [25], which defines small step operational semantics for NesC.

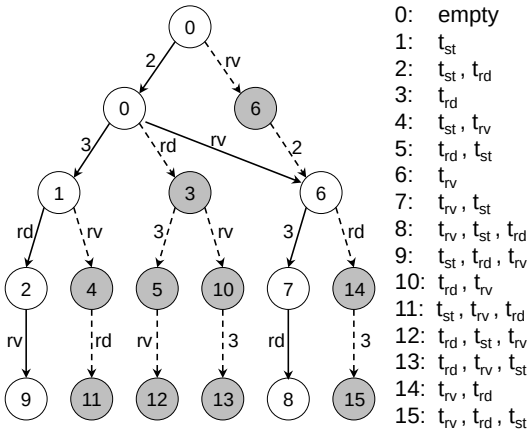
2.1 Interrupt-driven Sensors

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 not disabled. In TinyOS execution model [14], a task queue schedules tasks for execution, in FIFO order. 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 behaviors of devices are correct. 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. However, different orders of interrupt handler executions 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 is due to the assumption that devices work properly and thus hardware requests are assumed to be completed during 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. 2(a)) 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. 2(a)) 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. 2(a) illustrates a fragment of a NesC program, which involves messaging and sensing. The command call *Read.read*()/*Send.send*() invokes the corresponding command body that requests the sensing device/messaging device to read a data/to send a packet, which will later trigger the completion event *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, the state graph of event *Boot.booted* is shown in Fig. 2(b), 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 listed in the figure. For example in Fig. 2(b), initially, after executing *call Read.read*() (line 2) the task queue still remains empty, while after executing the interrupt handler *rv*, the completion task t_{rv} is enqueued and the task queue becomes $\langle t_{rv} \rangle$ (i.e., state 1).

```
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 }
```

(a) Example Code



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

Fig. 2. Interrupt-driven Sensor

2.2 Formal Definitions of Sensor Networks

The formal definitions of sensor networks 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; and P is a program composed by the running NesC program M and interrupting devices H, i.e., $P = M \triangle H$.*

A state C of $\mathcal{S}$ is a tuple (V, Q, B, P) where V is the current valuation of variable set A; Q is a sequence of tasks (i.e., the content of T); B is a packet (i.e., the content of R); and P is the program counter. In this work, we use $V(C)$,

$Q(C)$, $B(C)$ and $P(C)$ to denote the variable valuation, task queue, message buffer and program counter of a state C, respectively.

The transition system of $\mathcal{S}$ is defined as a tuple $\mathbb{T} = (\mathbb{C}, init, \rightarrow_s)$, where $\mathbb{C}$ is the set of all reachable states and $\rightarrow_s$ is the sensor transition relation. The transition relation $\rightarrow_s$ is formally defined through the operational semantics of NesC programming constructs [25]. A sensor transition t is defined as $C \xrightarrow{\alpha}_s C'$, where C and C' are the states before and after executing the action α. 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. $\sum_{\mathcal{S}}$ (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. $Tasks(\mathcal{S})$ (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 sensor network state $\mathcal{C}$ is defined as an ordered set of states $\{C_0, \cdots, C_n\}$ where C_i $(0 \leqslant i \leqslant n)$ is the state of $\mathcal{S}_i$, denoted as $\mathcal{C}[i]$. The sensor network transition system corresponding to $\mathcal{N}$ is a 3-tuple $\mathcal{T} = (\Gamma, init, \hookrightarrow)$ where Γ is the set of all reachable network states, $init = \{init_0, \cdots, init_n\}$ ($init_i$ is the initial state of $\mathcal{S}_i$) being the initial network state of $\mathcal{N}$, and $\hookrightarrow$ is the network transition relation. A sensor network transition $\tilde{T}$ is defined as $\mathcal{C} \xhookrightarrow{\alpha} \mathcal{C}'$ where $\mathcal{C}$ and $\mathcal{C}'$ are the network state before and after the transition, represented as $\mathcal{C}' = Ex(\mathcal{C}, \alpha)$. A network transition is generated either by a *local* sensor transition, e.g., updating a local variable or triggering an interrupt; or a *global* transition which involves network communication.

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 whereas non-post actions never access the task queue. For example, in Fig. 2(b), 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, like *rd* and *rv* at state 2, and line 3 and *rd* at state 1. This is because that t_{rd} only accesses variable *value* which is never accessed by t_{st} or t_{rv}. Therefore, it is important to detect the independence among actions inside a sensor, which is referred to as *local independence*.

From the view of the 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 affecting verification results. This observation leads to the

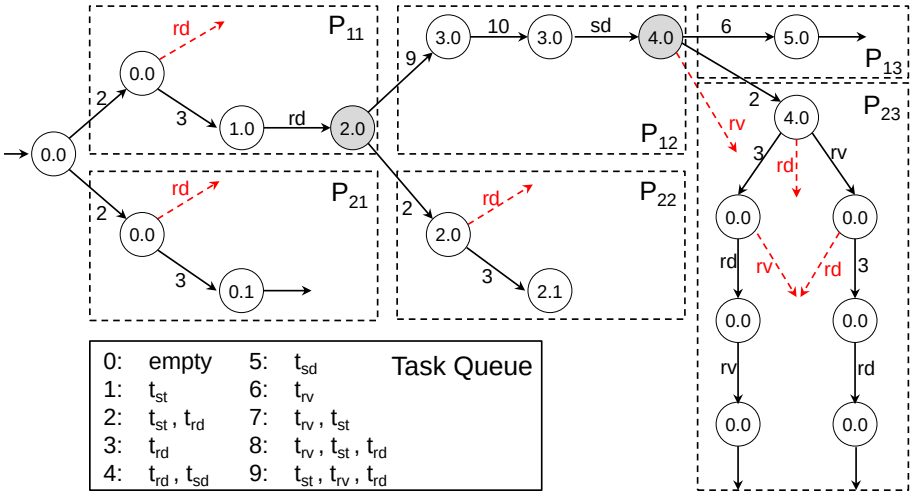


Fig. 3. Motivating Example: Two-level POR

independence analysis at 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. 2(a). Applying partial order reduction at both network and sensor levels, we are able to obtain a reduced state graph as shown in Fig. 3, where states are numbered with the task queues of both sensors, e.g., 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 two sensors is only allowed when necessary, like at the shadowed 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 used to prune unnecessary interleaving among local actions. Dashed arrows indicate pruned local actions. For example, rectangle p_{23} is constructed by removing all shadowed states and dashed transitions in Fig. 2(b).

3.1 Local Independence

In the following, we present the definitions of local and global independence and syntactic conditions which can be used to detect local and global independence. In a sensor model $\mathcal{S}$, an action may modify a variable or the task queue. Thus local independence is defined to describe independent actions according to their effect on the variables and the task queue.

Definition 3 (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. $Q(ex(ex(C, \alpha_1), \alpha_2)) \simeq Q(ex(ex(C, \alpha_2)), \alpha_1)$ (refer to Definition 6 for $\simeq$).

In the above definition, $=_v$ (referred to as *v-equal*) denotes that two 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 as $\alpha_1 \equiv_{VI} \alpha_2$. Let W_α and R_α be the set of variables written and only read by an action α, respectively.

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

Proof Suppose that $\alpha_1, \alpha_2 \in enable(C_0)$.Let $C_{12} = ex(ex(C_0, \alpha_1), \alpha_2)$ and $C_{21} = ex(ex(C_0, \alpha_2), \alpha_1)$. Since R_{α_1} are only read by α_1, trivially we have $R_{\alpha_1}(C_{12}) = R_{\alpha_1}(C_{21})$ (1). Assume that $C_{12} = C_0(W_{\alpha_1}/V_1', W_{\alpha_2}/V_2')$ and $C_{21} = C_0(W_{\alpha_2}/V_2'', W_{\alpha_1}/V_1'')$. Since V_1' and V_1'' are only dependent with $R_{\alpha_1} \cup W_{\alpha_1}$, with (1) and $W_{\alpha_1} \cap W_{\alpha_2} = \varnothing$, we can imply that $V_1' = V_1''$. Consequently, $W_{\alpha_1}(C_{12}) = W_{\alpha_1}(C_{21})$. Similarly we can prove that $W_{\alpha_2}(C_{12}) = W_{\alpha_2}(C_{21})$, and conclude that $V(C_{12}) = V(C_{21})$, i.e., $\alpha_1 \equiv_{VI} \alpha_2$. □

Lemma 1 shows that two actions are independent if the variables modified by one action are mutual exclusive with those accessed by the other. For example, by Lemma 1, $\alpha_{l6} \equiv_{VI} \alpha_{l13}$, where $\alpha_{l6}(\alpha_{l13})$ refers to an action executing the statement at line 6 (line 13) of Fig. 2(a).

Interrupt handlers might run in parallel to form different orders of tasks in the resultant 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. 2(a), $Ptask(t_{rv}) = \{t_{st}\}$, 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, $Ptask(t_{st}) = \{t_{sd}\}$.

Note that more tasks can be enqueued while an enqueued task is executing. We define $Rtask(t)$ to represent all 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. 2(a), since $Ptask(t_{sd}) = \varnothing$, we have $Rtask(t_{sd}) = \{t_{sd}\}$. Similarly, we obtain that $Rtask(t_{st}) = \{t_{st}, t_{sd}\}$ and $Rtask(t_{rv}) = \{t_{rv}, t_{st}, t_{sd}\}$. Let φ be a property and $R(\varphi, \mathcal{S})$ be the set of local variables of $\mathcal{S}$ accessed by φ. Let $\widehat{W}(t)$ be the set of variables modified by $Rtask(t)$. We say that t is φ-$\mathcal{S}$-safe, denoted as $t \in safe(\varphi, \mathcal{S})$ iff $(\widehat{W}(t) \cap R(\varphi, \mathcal{S})) = \varnothing$. In the following, we define local independence of tasks w.r.t. a certain property.

Definition 4 (Local Task Independence). *Let $t_i, t_j \in Tasks$ be two tasks. Given a property φ, t_i and t_j are said to be local-independent, denoted as $t_i \equiv_{TI}$*

t_j, *iff* $(t_i \in safe(\varphi,\mathcal{S}) \vee t_j \in safe(\varphi,\mathcal{S})) \ \wedge \ \forall\, t_i' \in Rtask(t_i), t_j' \in Rtask(t_j).\ t_i' \in safe(\varphi,\mathcal{S}) \ \wedge \ t_j' \in safe(\varphi,\mathcal{S}) \ \wedge \ \forall\, \alpha_i \in t_i', \alpha_j \in t_j'.\ \alpha_i \equiv_{VI} \alpha_j$.

Though interrupt handlers in asynchronous context and post statements in synchronous context both modify the task queue, we observe that task queues with different orders of tasks might be equivalent. In the following we define the independence 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 as* $Q_i \equiv_{SI} Q_j$, *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 $Swap(Q, i) = \langle q_0^\frown \cdots q_{i+1}^\frown q_i^\frown \cdots^\frown q_n\rangle$ denote the task sequence obtained by swapping the two sub-sequences (i.e., q_i and q_{i+1}) of Q, where $Q = \langle q_0^\frown q_1^\frown \cdots^\frown q_n\rangle$. Task sequence equivalence is defined in Definition 6. This observation is then adopted to reduce unnecessary interleaving among interrupt handlers and *post* actions.

Definition 6 (Task Sequence Equivalence). *Given two task sequences* Q *and* Q', *they are equivalent* ($Q \simeq Q'$) *iff* $Q^0 = Q \wedge \exists\, m \geqslant 0, Q^m = Q' \wedge (\forall\, k \in [0, m).\ \exists\, i_k.\ q_{i_k}^k \equiv_{SI} q_{i_k+1}^k \ \wedge \ Q^{k+1} = Swap(Q^k, i_k))$ *where* q_i^k *is the* i^{th} *subsequence 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 $Q \simeq Q'$. The relation $\simeq$ is reflexive, symmetric and transitive, as shown in Lemma 2.

Lemma 2 (Reflexivity, Symmetry and Transitivity of $\simeq$).
1. Reflexivity: $Q \simeq Q$; *(P1)*
2. Symmetry: $Q_1 \simeq Q_2 \Rightarrow Q_2 \simeq Q_1$; *(P2)*
3. Transitivity: $Q_1 \simeq Q_2 \wedge Q_2 \simeq Q_3 \Rightarrow Q_1 \simeq Q_3$. *(P3)*

Proof By Definition 6, $m = 0 \Rightarrow Q_1^m = Q_1^0 = Q$, thus $Q \simeq Q$ and $P1$ is proved.

Assume that $Q_1 \simeq Q_2$, if $Q_1 = Q_2$, by reflexivity it is trivial that $Q_2 \simeq Q_1$. Suppose that $Q_1 \neq Q_2$, then by Definition 6, we have $Q_1^0 = Q_1 \Rightarrow \exists\, m > 0, Q_1^m = Q_2 \wedge (\forall\, 1 \leq k \leq m, \exists\, i_k, q_{i_k}^{k-1} \equiv_{SI} q_{i_k+1}^{k-1} \wedge Q_1^k = Swap(Q_1^{k-1}, i_k))$ (1). Here q_i^k denotes the ith subsequences of Q_1^k. Let Q_2^k be the task sequence after swapping adjacent independent task sequences in Q_2 for k times, $\widetilde{q}_i^k$ denote the ith subsequence of Q_2^k. Assume $Q_2^0 = Q_2$ and let $Q_2^k = Swap(Q_2^{k-1}, i_{m-(k-1)})$. Then we have $Q_2^0 = Swap(Q_1^{m-1}, i_m) \wedge \widetilde{q}_{i_m}^0 = q_{i_m+1}^{m-1} \wedge \widetilde{q}_{i_m+1}^0 = q_{i_m}^{m-1}$, thus $\widetilde{q}_{i_m}^0 \equiv_{SI} \wedge \widetilde{q}_{i_m+1}^0$ and $Q_2^1 = Swap(Q_2^0, i_m) = Swap(Swap(Q_1^{m-1}, i_m), i_m) = Q_1^{m-1}$. Suppose that $Q_2^k = Q_1^{m-k} = Swap(Q_1^{m-k-1}, i_{m-k})$, then we have $\widetilde{q}_{i_{m-k}}^k = q_{i_{m-k}+1}^{m-k-1}$ and $\widetilde{q}_{i_{m-k}+1}^k = q_{i_{m-k}}^{m-k-1}$. By (1), we can infer that $q_{i_{m-k}}^{m-k-1} \equiv_{SI} q_{i_{m-k}+1}^{m-k-1}$. Therefore, $\widetilde{q}_{i_{m-k}}^k \equiv_{SI} \widetilde{q}_{i_{m-k}+1}^k$ and $Q_2^{k+1} = Swap(Q_2^k, i_{m-k}) = Swap(Q_1^{m-k}, i_{m-k}) =$

$Swap(Swap(Q_1^{m-(k+1)}, i_{m-((k+1)-1)}), i_{m-k}) = Q_1^{m-(k+1)}$. Inductively, $Q_2^m = Q_1^0 = Q_1$ and thus $Q_2 \simeq Q_1$. Consequently, $P2$ is proved.

Assume that $Q_1 \simeq Q_2 \wedge Q_2 \simeq Q_3$. If $Q_1 = Q_2$ or $Q_2 = Q_3$, then trivially $Q_1 \simeq Q_3$. Suppose that $Q_1 \neq Q_2$ and $Q_2 \neq Q_3$, by Definition 6, there exist $i_1, i_2, \cdots, i_m$ such that $Q_1^0 = Q_1 \wedge q_{i_k}^{k-1} \equiv_{SI} q_{i_k+1}^{k-1} \wedge Q_1^k = Swap(Q_1^{k-1}, i_k) \wedge Q_1^m = Q_2$ and $j_1, j_2, \cdots, j_n$ such that $Q_2^0 = Q_2 \wedge q_{j_k}^{k-1} \equiv_{SI} q_{j_k+1}^{k-1} \wedge Q_2^k = Swap(Q_2^{k-1}, j_k) \wedge Q_2^n = Q_3$. Again, let $\widetilde{Q}_1^0 = Q_1$, $v = m + n$, and $\forall\, 1 \leq k \leq v$, let $\widetilde{Q}_1^k = Swap(\widetilde{Q}_1^{k-1}, l_k)$ and $(1 \leq k \leq m \Rightarrow l_k = i_k) \wedge ((m+1) \leq k \leq (m+n) \Rightarrow l_k = j_k)$. Then we have $\widetilde{Q}_1^m = Q_1^m = Q_2$ and thus $\widetilde{Q}_1^{m+n} = Q_2^n = Q_3$. Therefore $Q_1 \simeq Q_3$ and $P3$ is proved. □

The following lemma shows another property of task sequence equivalence.

Lemma 3. $Q_1 \simeq Q_2 \Rightarrow Q_1^{\frown} Q' \simeq Q_2^{\frown} Q' \wedge Q'^{\frown} Q_1 \simeq Q'^{\frown} Q_2$.

Proof By Definition 6, there exists $m \geq 0$ such that $Q_0 = Q_1 \wedge Q^m = Q_2 \wedge \exists\, i_0, i_1, \cdots, i_k, \cdots, i_{m-1}.\ q_{i_k}^k \equiv_{SI} q_{i_k+1}^k \wedge Q^{k+1} = Swap(Q^k, i_k)$. Suppose that $Q_3 = Q_1^{\frown} Q'$ and $Q_4 = Q_2^{\frown} Q'$, then $Q_0 = Q_3 \wedge Q^m = Q_4 \wedge \forall\, i_0, i_1, \cdots, i_k, \cdots, i_{m-1}.\ q_{i_k}^k \equiv_{SI} q_{i_k+1}^k \wedge Q^{k+1} = Swap(Q^k, i_k)$. Thus $Q_3 \simeq Q_4$. Similarly, we can prove $Q'^{\frown} Q_1 \simeq Q'^{\frown} Q_2$. □

Given two 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 state sets $\mathbb{C}$, $\mathbb{C}'$ are said to be equivalent, denoted as $\mathbb{C} \asymp \mathbb{C}'$, 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 state which is the completion point of a previous task, i.e., a 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 $B(C) \neq \langle\rangle$ which is related to network communication is ignored here but will be covered in global independence analysis in Section 3.2.

Lemma 4. *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 states after executing all tasks of Q_i starting at state C_i. $Q \simeq Q' \Rightarrow exs(Q, C) \asymp exs(Q', C')$.*

Proof Let $Q = Q^0 = q_1^{0\frown} q_2^{0\frown} \cdots \frown q_n^0$. Since $Q \simeq Q'$, we can assume that Q' is obtained by applying m *Swap* actions on Q. Let $k \in [0, m)$, by Definition 6, we have $Q^m = Q'$ and for each k there exists i_k such that $q_{i_k}^k \equiv_{SI} q_{i_k+1}^k \wedge Q^{k+1} = Swap(Q^k, i_k))$. Let $C^k = (V, Q^k, \langle\rangle, \checkmark \triangle H)$ (this implies that $C^0 = C$ and $C^m = C'$), and $\widetilde{C}^k \in exs(Q^k, C^k)$. By Definition 5 and Lemma 1, $\forall\, \widetilde{C}^{k+1} \in exs(Q^{k+1}, C^{k+1}), V(\widetilde{C}^k) = V(\widetilde{C}^{k+1})$. Moreover, since all tasks and all interrupt handlers are completed after $exs(Q, C)$, thus $\forall\, \widetilde{C}^k \in exs(Q^k, C^k), \widetilde{C}^{k+1} \in exs(Q^{k+1}, C^{k+1}), P(\widetilde{C}^k) = P(\widetilde{C}^{k+1}) = \checkmark \triangle H$. Thus $\widetilde{C}^k =_v \widetilde{C}^{k+1}$.

Suppose that $q^k_{i_k} = \langle t^1_{i_k}, \cdots, t^i_{i_k} \rangle$ and $q^k_{i_k+1} = \langle t^1_{i_k+1}, \cdots, t^l_{i_k+1} \rangle$, and that $\widetilde{q}^k_{i_k} = \langle \widetilde{t}^1_{i_k}, \cdots, \widetilde{t}^x_{i_k} \rangle$ and $\widetilde{q}^k_{i_k+1} = \langle \widetilde{t}^1_{i_k+1}, \cdots, \widetilde{t}^y_{i_k+1} \rangle$ being the task sequences introduced by executing all tasks in $q^k_{i_k}$ and $q^k_{i_k+1}$ respectively. We remark that $\forall\, 1 \leq u \leq x, \widetilde{t}^u_{i_k} \in \cup^l_{v=1} Ptask(t^v_{i_k})$ and $\forall\, 1 \leq u \leq y, \widetilde{t}^u_{i_k+1} \in \cup^l_{v=1} Ptask(t^v_{i_k+1})$ (1) and $\forall\, 1 \leq u \leq x, 1 \leq v \leq y, \widetilde{t}^u_{i_k} \equiv_{TI} \widetilde{t}^v_{i_k+1}$(2). Then we have $Q(\widetilde{C}^k) = q_1^{k'} \frown \widetilde{q}^{k\frown}_{i_k} \widetilde{q}^{k\frown}_{i_k+1} q_1^{k''}$. According to the semantics that interrupts are allowed to interleave a task at any time, there exists $\widetilde{C}^{k+1}$ such that $Q(\widetilde{C}^{k+1}) = q_1^{k'} \frown \widetilde{q}^{k\frown}_{i_k+1} \widetilde{q}^{k\frown}_{i_k} q_1^{k''}$. If $\widetilde{q}^k_{i_k} = \langle\rangle$ or $\widetilde{q}^k_{i_k+1} = \langle\rangle$, then $Q(\widetilde{C}^k) = Q(\widetilde{C}^{k+1})$, and by the reflexivity of $\simeq$ we can obtain $Q(\widetilde{C}^k) \simeq Q(\widetilde{C}^{k+1})$. Now consider the case of $\widetilde{q}^k_{i_k} \neq \langle\rangle \wedge \widetilde{q}^k_{i_k+1} \neq \langle\rangle$. By (1), (2) and the definition of *Rtask*, $\cup^x_{u=1} Rtask(\widetilde{t}^u_{i_k}) \subseteq \cup^i_{u=1} Rtask(t^u_{i_k})$ and $\cup^y_{u=1} Rtask(\widetilde{t}^u_{i_k+1}) \subseteq \cup^l_{u=1} Rtask(t^u_{i_k+1})$. Thus $q^k_{i_k} \equiv_{SI} q^k_{i_k+1}$ implies that $\widetilde{q}^k_{i_k} \equiv_{SI} \widetilde{q}^k_{i_k+1}$ and $Q(\widetilde{C}^k) \simeq Q(\widetilde{C}^{k+1})$. Now we have proved that $\forall\, \widetilde{C}^k \in exs(Q^k, C^k), \exists\, \widetilde{C}^{k+1} \in exs(Q^{k+1}, C^{k+1}), \widetilde{C}^k =_v \widetilde{C}^{k+1} \wedge Q(\widetilde{C}^k) \simeq Q(\widetilde{C}^{k+1})$ (3).

Applying (3) to Q^0 for m times, with the transitivity of the $\simeq$ relation of task sequences, we can conclude that $\forall\, \widetilde{C}^0 \in exs(Q^0, C^0), \exists\, \widetilde{C}^m \in exs(Q^m, C^m), \widetilde{C}^0 =_v \widetilde{C}^m \wedge Q(\widetilde{C}^0) \simeq Q(\widetilde{C}^m)$. Equivalently, $\forall\, \widetilde{C} \in exs(Q, C), \exists\, \widetilde{C}' \in exs(Q', C'), \widetilde{C} \cong \widetilde{C}'$. Similarly, we can prove vice versa. □

Lemma 4 shows that executing two equivalent task sequences from *v-equal* states will always lead to equivalent sets of final states. 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 independency between actions can be obtained, as shown in Lemma 5.

Lemma 5. *Given a state C, $\alpha_1, \alpha_2 \in enable(C)$, $(\alpha_1 \equiv_{VI} \alpha_2 \wedge \forall\, t_1 \in ptsk(\alpha_1), t_2 \in ptsk(\alpha_2).\ t_1 \equiv_{TI} t_2) \Rightarrow \alpha_1 \equiv_{LI} \alpha_2$.*

Proof Since $\alpha_1 \equiv_{VI} \alpha_2$, we only need to prove the second condition in Definition 3. Suppose that $ex(C, \alpha_1), \alpha_2) = C_{12}$, $ex(C, \alpha_2), \alpha_1) = C_{21}$ and $Q(C) = Q_0$. then $Q(C_{12}) = Q_0^{\frown} q_1^{\frown} q_2$ and $Q(C_{21}) = Q_0^{\frown} q_2'^{\frown} q_1'$. Trivially $q_1, q_1' \subseteq ptsk(\alpha_1)$ and $q_2, q_2' \subseteq ptsk(\alpha_2)$. Since $\forall\, t_1 \in ptsk(\alpha_1), t_2 \in ptsk(\alpha_2).\ t_1 \equiv_{TI} t_2)$, $ptsk(\alpha_1) \cap ptsk(\alpha_2) = \varnothing$, and thus $q_1 = q_1'$ and $q_2 = q_2'$. Intuitively, $q_1^{\frown} q_2 \simeq q_2'^{\frown} q_1'$. Therefore, $Q(C_{12}) \simeq Q(C_{21})$. □

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 only visible to itself most of the time, except when it broadcasts a message to the network and fills in others' message buffers. In TinyOS, messages are queued for transmission and only when the previous transmission has completed, can a new message be sent successfully. This assures that during the execution of a task, at most one packet could be

successfully sent to the network. Thus, at network level, we explore the execution of each sensor by tasks, and only allow interleaving among sensors when the tasks are detected to involve network communication. Let $\mathcal{N}$ be a sensor network with n sensors $\mathcal{S}_1, \mathcal{S}_2 \cdots \mathcal{S}_n$. In the following, we define *global independence* of tasks. We use $EnableT(\mathcal{C})$ to denote the set of enabled tasks at network state $\mathcal{C}$. 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 final states after executing task t (and interrupt handlers caused by it) starting from $\mathcal{C}$. For two network 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 network state sets 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 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 as $t \subset_{RI} \mathcal{S}$, 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' tasks that perform a data transmission. Intuitively, if $t \in Sends(\mathcal{S})$, then interleaving among sensors is necessary. We say that t is a global-safe task of $\mathcal{S}$ iff $t \subset_{GI} \mathcal{S} \equiv t \notin Sends(\mathcal{S}) \wedge t \subset_{RI} \mathcal{S}$. If $t \not\subset_{GI} \mathcal{S}$, then t is global-unsafe. The following theorem indicates that a global-safe task is always global-dependent with any task of other sensors.

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

Proof Supposing that $t_2 \notin Sends(\mathcal{S}_j)$, since $t_1 \notin Sends(\mathcal{S}_i)$, we can prove immediately that $t_1 \equiv_{GI} t_2$. Suppose that $t_2 \in Sends(\mathcal{S}_j)$ and $\mathcal{S}_i$ is connected with $\mathcal{S}_j$. Thus after executing t_2, the message buffer of $\mathcal{S}_i$ might be modified. Suppose that $t_1, t_2 \in EnableT(\mathcal{C}_0)$ and $\mathcal{C}_0[i] = (V, \langle t_1 \rangle^{\frown} q_i)$, and let $\mathcal{C}_2 \in Ex(\mathcal{C}_0, t_2)$ and $\mathcal{C}_{21} \in Ex(\mathcal{C}_2, t_1)$. Let $\mathcal{C}_{21}[i] = (V_{12}^i, q_i^{\frown} \langle t_1' \cdots t_{rv} \cdots t_m' \rangle, \varnothing, \checkmark \triangle H)$, where t_{rv} is the task posted by packet arrival interrupt handler of $\mathcal{S}_i$ and for every $t_k'(1 \leq k \leq m)$ we have $t \in Ptask(t_1)$. It is immediately that there exists $\mathcal{C}_1 \in Ex(\mathcal{C}_0, t_1)$ such that there exists $\mathcal{C}_{12} \in Ex(\mathcal{C}_1, t_2)$, $\mathcal{C}_{12}[j] = \mathcal{C}_{21}[j]$, $\mathcal{C}_{12}[i] =_v \mathcal{C}_{21}[i]$ and $Q(\mathcal{C}_{12}[i]) = q_i^{\frown} \langle t_1', t_2' \cdots t_m', t_{rv} \rangle$. Intuitively, $B(\mathcal{C}_{12}[i]) \neq \varnothing$, since a packet arrival interrupt in $\mathcal{S}_i$ is triggered after t_2 is executed and the task t_{rv} is enqueued to be the last element. By the definition of $\subset_{RI}$ and Lemma 4, we have $Q(\mathcal{C}_{12}[i]) \simeq Q(\mathcal{C}_{21}[i])$, and thus $\mathcal{C}_{12}[i] \cong \mathcal{C}_{21}[i]$. It is obvious that for all $k(1 \leq k \leq n \wedge k \neq i, j$, we have $\mathcal{C}_{12}[i] = \mathcal{C}_{21}[i]$. Consequently, $\mathcal{C}_{21} \cong \mathcal{C}_{12}$. Similarly, we can prove vice versa. □

4 Sensor Network Cartesian Semantics

Cartesian POR was proposed by Gueta et. al. to reduce nondeterminism 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 an trace executing transitions only from the i^{th} process 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 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 $\mathit{Gprop}(\mathcal{N})$, or simply Gprop since $\mathcal{N}$ is clear in this section, be the set of *global* properties defined for $\mathcal{N}$. Given an action $\alpha \in \mathit{Tasks}(\mathcal{S})$ and a global property $\varphi \in \mathit{Gprop}$, α is said to be φ-safe, denoted as $\alpha \in \mathit{safe}(\varphi)$, iff $W_\alpha \cap R_\varphi = \varnothing$ where W_α is the set of variables written by α and R_φ is the set of variables accessed by φ. If $\alpha \notin \mathit{safe}(\varphi)$, then α is referred to as φ-unsafe.

4.1 Sensor Prefix

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

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

Let $p \in \mathit{Prefix}(\mathcal{S})$ and $p = (p_{tr}, br)$. We use $\mathit{trunk}(p)$ to denote the sub-prefix of p before branching prefixes (i.e., $\mathit{trunk}(p) = p_{tr}$); $\mathit{branch}(p)$ to denote the set of branching prefixes of p (i.e., $\mathit{branch}(p) = br$). In Fig. 3, the dashed rectangles p_{11}, p_{12} and p_{11} are prefixes of $\mathcal{S}_1$, and p_{21}, p_{22} and p_{21} are prefixes of $\mathcal{S}_2$. More specifically, $\mathit{trunck}(p_{23}) = \langle (4.0), \alpha_2, (4.0) \rangle$ and $\mathit{branch}(p_{23}) = \{\langle (4.0), \alpha_3, (4.1), \alpha_{rd}, \cdots \rangle, \langle (4.0), \alpha_{rv}, (4.6), \alpha_3 \rangle\}$. Given a prefix $p \in \mathit{Prefix}(\mathcal{S})$, the following definitions are defined for our POR approach.

- The set of states in p including those in its branching prefixes: $\mathit{states}(p) = \{\mathcal{C}_0, \cdots, \mathcal{C}_m\} \cup (\cup_{sp \in branch(p)} \mathit{states}(sp))$.
- The set of *leaf* prefixes of $\mathcal{S}$: $\mathit{LeafPrefix}(\mathcal{S}) = \{lp \mid \forall\, lp \in \mathit{Prefix}(\mathcal{S}).\ \mathit{branch}(lp) = \varnothing\}$. Given $lp \in \mathit{LeafPrefix}(\mathcal{S})$, $\widehat{last}(lp)$ denote the last state of lp.
- The set of *tree* prefixes of $\mathcal{S}$: $\mathit{TreePrefix}(\mathcal{S}) = \mathit{Prefix}(\mathcal{S}) - \mathit{LeafPrefix}(\mathcal{S})$.
- The set of leaf prefixes of p: $p \in \mathit{LeafPrefix}(\mathcal{S}) \Rightarrow \mathit{leaf}(p) = p \ \wedge \ p \in \mathit{TreePrefix}(\mathcal{S}) \Rightarrow \mathit{leaf}(p) = \cup_{sp \in branch(p)} \mathit{leaf}(sp)$.

– The set of final states of p: $p \in \mathit{LeafPrefix}(\mathcal{S}) \Rightarrow \mathit{last}(p) = \{\widehat{\mathit{last}}(p)\} \ \wedge \ p \in \mathit{TreePrefix}(\mathcal{S}) \Rightarrow \mathit{last}(p) = \cup_{bp \in branch(p)} \mathit{last}(bp)$.
– Subsequent prefixes $\sqsupset$: $\forall\, lp \in \mathit{LeafPrefix}(\mathcal{S}).\ lp \sqsupset p \equiv \widehat{\mathit{last}}(lp) = \mathit{first}(p)$.
– Combination 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 $\mathit{tasks}(p)$ ($\mathit{acts}(p)$) in a similar way to $\mathit{states}(p)$ to denote the set of all tasks (actions) executed in p. Moreover, $\mathit{lastT}(p)$ ($\mathit{lastAct}(p)$) are defined in a similar way to $\mathit{last}(p)$ to denote the set of last tasks (actions) executed in p.

4.2 Sensor Network Cartesian Vector

Definition 9 (Sensor Network Cartesian Vector).
Given a global property $\varphi \in \mathit{Gprop}$, a vector $(p_1, \cdots, p_n) \in \mathit{Prefix}^n$ is a sensor network cartesian vector for $\mathcal{N}$ w.r.t. φ from a network state $\mathcal{C}$ if the following conditions hold, where $1 \leq i \leq n$:

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

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. In Fig. 3, if φ is not given, then (p_{11}, p_{21}) is a valid SNCV from the initial state.

To generate a sensor network cartesian vector for any explored state, we assume the existence of an cartesian function ϕ: $\Gamma \times \mathit{Prefix}^n$ such that, for every $\mathcal{C} \in \Gamma$, $\phi(\mathcal{C})$ is a cartesian vector from $\mathcal{C}$. Given a cartesian function ϕ, we can build a cartesian semantics that uses ϕ as a guide for execution. When the cartesian semantics starts the execution from a state $\mathcal{C}$ it selects a prefix p from the vector $\phi(\mathcal{C})$ and executes the transitions of p. When the semantics reaches a state $\mathcal{C}' \in \mathit{last}(p)$, it executes the function ϕ again from $\mathcal{C}'$.

The cartesian semantics generated by ϕ is formalized as two binary relations $\rightarrow_\phi$ and $\Rightarrow_\phi$ on states, where $\rightarrow_\phi$ relates final states at the end of prefixes and is transitively closed, and $\Rightarrow_\phi$ extends $\rightarrow_\phi$ to also include intermediate states. In the following, we define the corresponding inference rules of $\rightarrow_\phi$ and $\Rightarrow_\phi$.

$$\frac{}{\mathcal{C} \rightarrow_\phi \mathcal{C}} \quad [\ \textit{reflexivity}\ ]$$

$$\frac{i \in [1, n],\ \exists\, p \in \mathit{Prefix}(\mathcal{C}, S_i)\ .\ \mathcal{C}' \in \mathit{last}(p)}{\mathcal{C} \rightarrow_\phi \mathcal{C}'} \quad [\ \textit{basis}\ ]$$

$$\frac{\mathcal{C} \rightarrow_\phi \mathcal{C}',\ \mathcal{C}' \rightarrow_\phi \mathcal{C}''}{\mathcal{C} \rightarrow_\phi \mathcal{C}''} \quad [\ \textit{transitivity}\ ]$$

Algorithm 1 State Space Generation

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

```
1: list ← ∅                                   6:    for all i ← 1 to n do
2: if Next(p, C) ≠ ∅ then                     7:       list ← list ∪ {Next(scv[i], C)}
3:    list ← Next(p, C)                       8:    end for
4: else                                       9: end if
5:    scv ← GetNewCV(C, φ)                   10: return list
```

$$\frac{}{\mathcal{C} \Rightarrow_\phi \mathcal{C}} \quad [\ \textit{reflexivity}\ ]$$

$$\frac{i \in [1,n],\ \exists\, p \in \textit{Prefix}(\mathcal{C}, S_i)\ .\ \mathcal{C}' \in \textit{states}(p)}{\mathcal{C} \Rightarrow_\phi \mathcal{C}'} \quad [\ \textit{basis}\ ]$$

$$\frac{\mathcal{C} \rightarrow_\phi \mathcal{C}',\ \mathcal{C}' \Rightarrow_\phi \mathcal{C}''}{\mathcal{C} \Rightarrow_\phi \mathcal{C}''} \quad [\ \textit{pseudo} - \textit{transitivity}\ ]$$

5 Two-level POR Algorithm

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] to achieve maximum reduction. In this section, we use $\mathcal{N}$ to denote a sensor network with n sensors: $\mathcal{S}_1, \cdots, \mathcal{S}_n$, and $\mathcal{C}$ denotes a network state of $\mathcal{N}$.

5.1 Algorithms

Given a network state $\mathcal{C}$, a prefix p ($\mathcal{C} \in \textit{states}(p)$) 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, $\textit{GetNewCV}(\mathcal{C}, \varphi)$ generates a new SNCV from $\mathcal{C}$, which will be explained later. The relation $\textit{Next} : \textit{Prefix}(\mathcal{S}) \times \Gamma \rightarrow \mathbb{P}(\Gamma)$ traverses a prefix to find a set of successors of $\mathcal{C}$. Formally, $\textit{Next}(p, \mathcal{C}) = \{\mathcal{C}' \mid \exists\, \alpha \in \textit{acts}(p), \mathcal{C} \overset{\alpha}{\hookrightarrow} \mathcal{C}'\}$. The function *CombineTree* extends a leaf prefix with another prefix as its branch, defined as $\textit{CombineTree}(lp \in \textit{LeafPrefix}(\mathcal{S}), sp \in \textit{Prefix}(\mathcal{S}))$. Formally, if $lp \sqsupset sp$, after executing $\textit{CombineTree}(lp, sp)$, we have $lp' = (lp, \{sp\})$. We remark that $\textit{CombineTree}(lp, sp)$ has a side effect in lp by updating it with the resultant prefix of the combination.

The method $\textit{GetNewCV}(\mathcal{C}, \varphi)$ generates an SNCV from $\mathcal{C}$ w.r.t. φ, as shown in Algorithm 2, and is in fact a ϕ function as discussed in Section 4.2. 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

Algorithm 2 Sensor Network Cartesian Vector Generation

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

```
1: scv ← (⟨⟩, ··· , ⟨⟩)   (n components)
2: for all S_i ∈ 𝒩 do
3:    visited ← {𝒞}
4:    workingLeaf ← ∅
5:    p_i ← GetPrefix(S_i, 𝒞, φ)
6:    for all lp ∈ leaf(p_i) do
7:       if Extensible(lp, S_i, φ) and
         last^(lp) ∉ visited then
8:          workingLeaf.Push(lp)
9:          visited = visited ∪ last^(lp)
10:      end if
11:   end for
12:   while workingLeaf ≠ ∅ do
13:      p_k ← workingLeaf.Pop()
14:      visited ← visited ∪ {last^(p_k)}
15:      p'_k ← GetPrefix(S_i, last^(p_k), φ)
16:      ConcatTree(p_k, p'_k)
17:      for all lp ∈ leaf(p'_k) do
18:         if Extensible(lp, S_i, φ) and
            last^(lp) ∉ visited then
19:            workingLeaf.Push(lp)
20:         end if
21:      end for
22:   end while
23:   scv[i] ← p_i
24: end for
25: return scv
```

Extendable : $Prefix(\mathcal{S}) \times \{\mathcal{S}_1, \cdots, \mathcal{S}_n\} \times Gprop \rightarrow \{True, False\}$ is defined as $Extendable(p, \mathcal{S}, \varphi) \equiv \forall\, \alpha \in lastAct(p).\ \alpha \notin Sends(\mathcal{S}) \ \wedge \ \alpha \in safe(\varphi)$. In other words, a prefix is further extended (lines 15 to 21) only if it has not executed a global-unsafe task or a φ-unsafe 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. The i^{th} element of the sensor cartesian vector *scv* ($scv[i]$) is initialized as p_i, which is the prefix obtained by $GetPrefix(\mathcal{S}_i, \mathcal{C}, \varphi)$. And p_i is then extended by recursively combining each of its leaf prefixes with a new prefix obtained by *GetPrefix*, as shown by lines 12 to 22.

Algorithm 3 shows how a sensor establishes a prefix from $\mathcal{C}$ w.r.t. φ. The function $ExecuteTask(t, p, \varphi, \mathcal{C}s)$ 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 5. The details of *ExecuteTask* is shown in Algorithm. 4. A persistent set approach has been implemented in both *ExecuteTask* and *RunInterrupts* to constrain interleaving to happen only between local-dependent actions, the details of which could be found in Algorithms 5 and 6.

5.2 Correctness

In this section, we show that the above POR algorithms work properly and are sound for model checking global properties and LTL-X properties. Lemmas 6 and 7 describe the correctness of Algorithm. 4 and 5, respectively.

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 *ter-*

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

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**

minates or a φ-unsafe action or a loop is encountered. □

Lemma 7. *Given a network state $\mathcal{C}$ where $\mathcal{C}[i] = (V, Q, B, \checkmark \triangle H)$, RunInterrupts $(\mathcal{C}, \mathcal{S}_i.GetInterrupts(\mathcal{C}'))$ terminates and returns a valid prefix of $\mathcal{S}_i$.* □

Based on Lemma 4 and 5, 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. By

Algorithm 5 Interleaving Interrupts

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

```
1: if itrs ← ∅ then
2:     return ⟨⟩
3: end if
4: p ← ⟨C⟩
5: {pis is the persistent set of itrs}
6: pis ← GetPerSet(itrs, C, S)
7: {interleave dependent actions}
8: for all α ∈ pis do
9:     C' ← ex(C, α)
10:    lp ← ⟨C, α, C'⟩
11:    setPfx(C', lp)
12:    {only allow interleaving if α is not a post}
13:    if α ∈ ∑^iq then
14:        s ← RunItrs(C', pis − {α}, S)
15:        lp ← (lp, {s})
16:    end if
17:    {add lp as a new branch to p}
18:    p ← (tr(p), br(p) ∪ {lp})
19: end for
20: return p
```

Algorithm 6 Persistent Set

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

```
1: if ∃α' ∈ itrs. α ∉ itrQ(S) then
2:     α = α'
3: else
4:     α ∈ itrs
5: end if
6: pset ← {α}
7: work ← {α}
8: while work ≠ ∅ do
9:     α ← work.Pop()
10:    αs ← DepActions(α, itrs − pset) {Depedent actions of α}
11:    pset = pset ∪ αs
12:    work = work ∪ αs
13: end while
14: return pset
```

Lemma 7, p'_i is a valid prefix of $\mathcal{S}$. 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 is combined with p_i and no longer a leaf prefix. By Lemma 7, 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 7, both lines 3 and 8 terminates. Further, we assume that variables are finite-domained, and therefore 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. $p_i \in \mathit{Prefix}(\mathcal{S}_i, \mathcal{C})$;

2. $\mathit{workingPrefix} = \{p \in \mathit{leaf}(p_i) \mid \mathit{Extendable}(p, \mathcal{S}_i, \varphi) \wedge \widehat{\mathit{last}}(p) \notin \mathit{visited}\}$.

By line 5, it is immediately true that $\mathit{first}(p_i) = \mathcal{C}$, and since the function *CombineTree* never changes the first state of a given prefix, $\mathit{first}(p_i) = \mathcal{C}$ holds for all iterations. Since p_i is extended by executing $\mathit{GetPrefix}(\mathcal{S}_i, \widehat{\mathit{last}}(p_i), \varphi)$, which only executes actions of $\mathcal{S}_i$, thus $p_i \in \mathit{Prefix}(\mathcal{S}_i)$ always holds. Intuitively, condition 1 holds for all iterations. In the following we prove condition 2 by induction.

At the first iteration, by lines 6 to 11, we can immediately obtain that $\mathit{workingPrefix} = \{lp \in \mathit{leaf}(p_i) \mid \mathit{Extendable}(lp, \mathcal{S}_i, \varphi) \wedge \widehat{\mathit{last}}(lp) \notin \mathit{visited}\}$ and condition 2 holds. Suppose that at the beginning of the m^{th} iteration, condition 2 holds with $\mathit{workingLeaf} = w_m$, $p_i = p_m$. After executing line 13, we can obtain that $\mathit{workingPrefix} = w_m - \{p_k\}$. By lines 15 to 21, $\mathit{wokingPrefix} = w_m - \{p_k\} \cup \{lp \in \mathit{leaf}(p'_k) \mid \mathit{Extendable}(lp, \mathcal{S}_i, \varphi) \wedge \widehat{\mathit{last}}(lp) \notin \mathit{visited}\}$ (1). Let $\widehat{p}_k$ be the new value of p_k after executing line 16, by the definition of *CombineTree*, we have $\widehat{p}_k = (p_k, p'_k)$ and thus $\mathit{leaf}(\widehat{p}_k) = \mathit{leaf}(p'_k)$ (2). Consequently, we have $\mathit{leaf}(p_i) = \mathit{leaf}(p_m) - \{p_k\} \cup \mathit{leaf}(\widehat{p}_k)$, since the leaf prefix p_k has been extended to be a tree prefix $\widehat{p}_k$. Since $w_m = \{p \in \mathit{leaf}(p_m) \mid \mathit{Extendable}(p, \mathcal{S}_i, \varphi) \wedge \widehat{\mathit{last}}(p) \notin \mathit{visited}\}$, with (1) and (2), we can obtain that condition 2 holds at the beginning of the $(m+1)^{th}$ iteration, condition 2 holds. Therefore, we can conclude that $\forall\, t \in \mathit{tasks}(p_i), \alpha \in \mathit{acts}(p_i).\ t \not\subset_{GI} \mathcal{S}_i \Rightarrow t \in \mathit{LastT}(p_i) \wedge \alpha \notin \mathit{safe}(\varphi) \Rightarrow \alpha \in \mathit{lastAct}(p_i)$ holds when the while loop terminates. Thus the cartesian vector generated by Algorithm 2 is a valid sensor cartesian vector.

For termination, we assume that all variables are finite-domained and thus the state space of each sensor is finite. On one hand, the function $\mathit{GetPrefix}(\mathcal{S}, \mathcal{C}, \varphi)$ always terminates and returns a valid prefix, which has been proved in Theorem 6. On the other hand, Algorithm 2 uses *visited* to store each state that has been used to generate a new prefix, and by lines 7 and 18 a state is used at most once to generate new prefix. Thus termination is guaranteed. □

Let φ be a property, ψ be the set of propositions belonging to φ, and $L(\mathcal{C})$ 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 $\sigma' = \mathcal{C}'_0, \alpha'_0, \mathcal{C}'_1, \alpha'_1, \cdots, \mathcal{C}'_i, \alpha'_i, \cdots$, we then define two stuttering equivalent traces w.r.t. φ, denoted as , in the following definition.

Definition 10 (Stuttering Equivalent Traces). *Two traces $\sigma = \mathcal{C}_0, \alpha_0, \mathcal{C}_1, \alpha_1, \cdots, \mathcal{C}_i, \alpha_i, \cdots$ and $\sigma' = \mathcal{C}'_0, \alpha'_0, \mathcal{C}'_1, \alpha'_1, \cdots, \mathcal{C}'_i, \alpha'_i, \cdots$ are said to be stuttering equivalent w.r.t. φ iff for every $M = \{m_0, m_1, \cdots, m_i\}$ $(i \geq 0)$ such that $m_0 = 0$, for all $k \geq 0$, $m_{k+1} = m_k + n_k \wedge L(\mathcal{C}_{m_k}) = L(\mathcal{C}_{m_k+1}) = \cdots = L(\mathcal{C}_{m_k+(n_k-1)}) \neq L(\mathcal{C}_{m_{k+1}})$, there exists $P = \{p_0, p_1, \cdots, p_i\}$ such that $p_0 = 0$, $p_{k+1} = p_k + q_k \wedge L(\mathcal{C}_{m_k}) = L(\mathcal{C}'_{p_k}) = L(\mathcal{C}'_{p_k+1}) = \cdots = L(\mathcal{C}'_{p_k+(q_k-1)}) \wedge L(\mathcal{C}'_{p_k+1}) = L(\mathcal{C}_{m_{k+1}})$ and vice versa.*

Given a local state $C = (V, Q, B, P)$ for a certain sensor $\mathcal{S}$, let $\mathit{trExs}(C, Q)$ be the set of all possible traces after executing all tasks in Q. In the following, we

define task sequences that generate stuttering equivalent traces. Given two local states $C = (V, Q, B, P)$ and $C' = (V, Q', B, P)$, Q and Q' are stuttering equivalent w.r.t. ψ ($Q \simeq_{st_\psi} Q'$) iff $\forall\, \sigma \in trExs(C, Q).\ \exists\, \sigma' \in trExs(C', Q').\ \sigma \equiv_{st_\psi} \sigma'$ and vice versa.

The relation $\sigma \equiv_{st_\varphi}$ is transitive [1], as indicated by Lemma 8.

Lemma 8. *Given three task sequences Q, Q', Q'' such that $Q \simeq_{st_\varphi} Q'$ and $Q' \simeq_{st_\varphi} Q''$ then $Q \simeq_{st_\varphi} Q''$.*

Proof Immediate by definition of stuttering equivalence of task sequences and by the transitivity of stuttering. □

Lemma 9. *Given two task sequences Q_1 and Q_2, $Q_1 \simeq Q_2$ w.r.t. $\varphi \Rightarrow Q_1 \equiv_{st_\psi} Q_2$.* □

Lemma 10 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.

Lemma 10. *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.* □

Proof Since C, C' have the same valuation of variables V then $L(C) = L(C')$ (1). Interruptions do not modify the values of variables V in an state C, it holds for σ and σ' that for all j $0 \leq j \leq n$ such that $\alpha_j \in itrQ(S)$, C_j,α_j,C_{j+1} then $L(C_j) = L(C_{j+1})$ (2). Given $\sigma = C$,α_0,$\cdots$,C_r,α_r,C_{r+1},$\cdots$,α_{n-1},C_n By Algorithm 4, *ExecuteTask* returns a trace $\sigma' = C'$,α'_0,$\cdots$,C_s,α_s,C_{s+1},$\cdots$,α'_{n-1},C'_n such that for every $\alpha_r \in \sigma$, $\alpha_r \in t$ then there exist $\alpha_s \in \sigma$, $\alpha_s \in t$ and $\alpha_r = \alpha_s$ and for all $r_< \ 0 \leq r_< < r$ $\alpha_{r_<} \in \sigma$, $\alpha_{r_<} \in t$ there exist $\alpha_{s_<} \in \sigma'$ such that $\alpha_{r_<} = \alpha_{s_<}$, and for all $r_> \ n-1 \leq r_> > r$ $\alpha_{r_>} \in \sigma$, $\alpha_{r_>} \in t$ there exist $\alpha_{s_>} \in \sigma'$, $\alpha_{s_>} \in t$ such that $\alpha_{r_>} = \alpha_{s_>}$ (3).Then by (1) and (2) $L(C_r) = L(C_s)$ and $L(C_{r+1}) = L(C_{s+1})$ therefore for all traces σ' from a prefix p returned by $GetPrefix(C')$ holds that $\sigma \equiv_{st_\varphi} \sigma'$.

On the other hand, let $\sigma_p = \mathcal{C} = c_0$, α_{p_0},C_{p_0},$\cdots$,$\alpha_{p_{n-1}}$,α_{p_n}, $C_{p_{n+1}}$ with $C_{p_{n+1}} = (V, Q^{\frown} Q_p, B, P_p)$ where σ_p is the result of removing all the actions that do not modify the task queue, and $Q_p = tsk(\alpha_{p_0})^{\frown} \cdots^{\frown} tsk(\alpha_{p_n})$ therefore $Q_p = Q_\sigma$(4). By Algorithm 4 and as it is shown in (3) all the *Post* actions α_p in σ belong to all the traces σ' from a prefix p that $ExecuteTask(C')$ returns, and every α_p is in the same relative positions w.r.t. other α'_p such that $\alpha'_p \neq \alpha_p$. Since $C' = (V, Q', B, P)$ such that V, B, P are the same as in C then for all the actions $\alpha_{int} \in \sigma$ such that $\alpha_{int} \in itrQ(S)$ and for all $\sigma' \in traces(p)$ there exist an action $\alpha'_{int} \in \sigma'$ such that $\alpha_{int} = \alpha'_{int}$. For any α_γ, where α_γ is a post action, let $ints_\gamma = \{\alpha_{\gamma_0}, \cdots \alpha_{\gamma_n}\}$ where for all $i, 0 \leq i \leq n$ $\alpha_{\gamma_i} \in itrQ$ and for all $\alpha_{p_k} \in \sigma_p$ such that $p_k < \gamma_0$ then there exist a Post action α'_γ such that

$\gamma' < \gamma$ and $p_k < \gamma' < \gamma_0$, and for all $\alpha'_{p_k} \in itrQ$ such that $p'_k > \gamma_n$ then there exist a Post action α''_γ such that $\gamma'' > \gamma$ and $p'_k > \gamma'' > \gamma_n$. Let consider the set $dep\{\alpha\})$ which is defined as the fixed point of $dep(X) = X' = \{\alpha_j \in ints_\gamma. \forall\, \alpha'_j \in s, \alpha_j \not\equiv_{TI} \alpha'_j \cup dep(X')$. By induction in the number of Post actions in σ_p $Q_\sigma \simeq_{st} Q'_\sigma$ is proven:

Base Case The number of Post actions in σ_p is zero. Let $\Omega_{dep} = \{\alpha_{\varphi_0}, \cdots, \alpha_{\varphi_l}\}$ the set of actions such that for some $t \in Rtask(tsk(\alpha_{\varphi_i}))$, $t \notin safe(\varphi, \mathcal{S})$. Since by Definition 4 for all the actions $\alpha_s \in \sigma_p . \alpha_s \notin \Omega_{dep}$ $tsk(\alpha_s) \in safe(\varphi, \mathcal{S})$ then the valuation of the truth values of ψ while executing $tsk(\alpha_s)$ will not vary and for all trace from t $\sigma_t = C_{t_0}, \alpha_{t_0}, \cdots, \alpha_{t_m}, C_{t_{m+1}}$ for all i $0 \leq i \leq m$ $L(C_{t_i}) = L(C_{t_{i+1}})$ (1). Let $\sigma_\varphi = C = C_{\varphi_0}, \alpha_{\varphi_0}, \cdots, \alpha_{\varphi_l}, C_{\varphi_{l+1}} = (V, Q^\cap Q'_p, B, P_\sigma)$ with $Q'_p = tsk(\alpha_{\varphi_0})^\cap \cdots^\cap tsk(\alpha_{\varphi_n})$ and by (1) $Q'_p \simeq_{st} Q_p$. By Algorithms 4, 5 and 6 $GetPrefix(C')$ returns a prefix p that contains a trace $\sigma' = C', \alpha'_0, \cdots, \alpha'_j, C_{\alpha'_{j+1}}, \sigma_\varphi, \cdots, \alpha'_n, C'_{n+1} = (V, Q^\cap Q'_\sigma, B, P_p)$ with $C_{\sigma_\varphi} = (V, Q^\cap Q_{\sigma_\varphi}, B, P_\gamma)$ where for all i, $0 \leq i \leq n$ α'_n does not enqueue any task which modifies the valuation value of ψ in any state and $Q_{\sigma_\varphi} \simeq_{st} Q'_\sigma$. By lemma 8 $Q''_p \simeq_{st} Q'_p$ and $Q''_p \simeq_{st} Q_p$ then $Q_\sigma \simeq_{st} Q'_\sigma$.

Induction Step Let suppose that the number of Post actions in σ_p is n and $Q_\sigma \simeq_{st} Q'_\sigma$. Suppose that the number of Post actions is $n+1$. Let $\alpha_\gamma = Post(t)$ the first Post operation in $\sigma_p = C = C_0, \alpha_{p_0}, C_{p_0}, \cdots, \alpha_\gamma, C_\gamma, \cdots, \alpha_{p_{m-1}}, C_{p_m}$ and $C_\gamma = (V, Q^\cap Q_\gamma, B, P_\gamma)$ and $Q_\gamma = tsk(\alpha_{\gamma_0})^\cap \cdots^\cap tsk(\alpha_{\gamma_n})^\cap t$. Let $\Omega_{dep} = dep(\alpha_\gamma) \cup \alpha_\gamma$, as explained above all the enqueued tasks $t = tsk(\alpha_{int})$ such that $\alpha_{int} \in ints_\gamma$ and $\alpha_{int} \notin \Omega_{dep}$ do not alter the valuation of the propositions of ψ. Let $\sigma_\varphi = C = C_{\varphi_0}, \alpha_{\varphi_0}, \cdots, \alpha_{\varphi_l}, \mathcal{C}_{\varphi_l}, \alpha_\gamma, C'_\gamma = (V, Q^\cap Q'_\gamma, B, P_\varphi)$ such that for all i, $0 \leq i \leq l$ $\alpha_{\varphi_i} \in ints$ and $\alpha_{\varphi_i} \in \Omega_{dep}$ that is, σ_φ is composed by actions in $ints$ without the independent tasks and $Q'_\gamma = tsk(\alpha_{\varphi_0})^\cap \cdots^\cap \alpha_{\varphi_{l-1}})^\cap t$. Therefore $Q'_\gamma \simeq_{st} Q_\gamma$. By Algorithms 4, 5 and 6 $GetPrefix(C')$ returns a prefix p that contains a trace $\sigma' = C', \alpha'_0, \cdots, \alpha'_j, \mathcal{C}'_{\alpha'_j}, \sigma_\varphi, \cdots, \alpha'_{n-1}, \mathcal{C}'_n$ where for all i, $0 \leq i \leq j$ α'_i does not enqueue any task which modifies the valuation value of ψ in any state. Let $\sigma'_p = C', \alpha'_{p_0}, \cdots, \sigma_\varphi, C_{\sigma_\varphi}, \cdots, \alpha'_{p_{q-1}}, C'_{p_q}$ the trace containing only actions from σ' that modifies the task queue with $C_{\sigma_\varphi} = (V, Q^\cap Q''_\gamma, B, P_\gamma)$. Therefore $Q''_\gamma \simeq_{st} Q'_\gamma$ and $Q''_\gamma \simeq_{st} Q_\gamma$. By the I.H. $\sigma_\gamma = \mathcal{C}_\gamma, \alpha_{\gamma+1}, \cdots, \alpha_{p_{m-1}}$, $C_{p_m} = (V_p, Q^\cap Q'^\cap_\gamma Q_{\gamma_0}, B_p, P_p)$ there exist a sequence of tasks $\sigma'_\gamma = C_{\sigma_\varphi}, \alpha_{p_0}, \cdots, \alpha'_{p_{m-1}}$, $C'_{p_m} = (V_p, Q^\cap Q''^\cap_\gamma Q'_{\gamma_0}, B_p, P_p)$ where $\cap Q'^\cap_\gamma Q_{\gamma_0} = Q_\sigma$, $Q''^\cap_\gamma Q'_{\gamma_0} = Q'_\sigma$ such that $Q'_{\gamma_0} \simeq_{st} Q_{\gamma_0}$. By By lemma 8 $Q_\sigma = Q'^\cap_\gamma Q_{\gamma_0} \simeq_{st} Q'_\sigma = Q''^\cap_\gamma Q'_{\gamma_0}$ and $Q_\sigma \simeq_{st} Q'_\sigma$ □

Lemma 11. *Given a property φ, two configurations $\mathcal{C} = \{C_0, \cdots, C_{ns}\} \in \mathcal{N}$, and $\mathcal{C}' = \{C'_0, \cdots, C'_{ns}\} \in \mathcal{N}$, and a sensor $\mathcal{S}_i$, then for any trace $\sigma = C_i, \alpha_0, \cdots, \alpha_{n-1}, C_n = (V_\sigma, Q_\sigma, B_\sigma, P_\sigma)$ where for all j $0 \leq j < n$ $\alpha_j \in \Sigma_{\mathcal{S}_i}$ and α_{n-1} holds some of the conditions 2., 3. from Definition 9 then Algorithm 2 returns a SNCV $sncv = (p_0, \cdots, p_{ns})$ such that there exist a trace σ' in p_i, $\sigma' = c' = c'_0, \alpha'_0, \cdots, \alpha'_i, \cdots, \alpha'_{n-1}, c'_n$, $c'_n = (V_\sigma, Q'_\sigma, B_\sigma, P_\sigma)$ where $\sigma \equiv_{st_\varphi} \sigma'$ and $Q_\sigma \simeq_{st} Q'_\sigma$.*

Proof By induction in the number of different tasks *ntasks* in σ:

Base Case $ntasks = 1$. Let $\sigma = C_i,\alpha_0,\cdots,\alpha_{n-1},C_n = (V_\sigma, Q_\sigma, B_\sigma, P_\sigma)$. Let p_i the prefix returned by $\sigma' = c' = c'_0,\alpha'_0,\cdots,\alpha'_i,\cdots,\alpha'_{n-1},c'_n$, $c'_n = (V_\sigma, Q'_\sigma, B_\sigma, P_\sigma)$. Immediately, by Lemma 10 $GetPrefix(C'_0)$ returns a prefix p such that contains a trace $\sigma'_i = c' = c'_0,\alpha'_0,\cdots,\alpha'_i,\cdots,\alpha'_{n-1},c'_n$, $c'_n = (V_\sigma, Q'_\sigma, B_\sigma, P_\sigma)$ where $\sigma \equiv_{st_\varphi} \sigma'$ and $Q_\sigma \simeq_{st} Q'_\sigma$.

Induction Step Let suppose that the hypothesis holds for σ containing n tasks. Given a trace σ containing actions from $n+1$ different tasks $\sigma = C_0,\alpha_0,\cdots,\alpha_i,C_{i+1},\cdots,\alpha_{n-1},C_n = (V_\sigma, Q_\sigma, B_\sigma, P_\sigma)$ let α_i an action such that for all j $0 \leq j < i$ there exist $t_j \in Tasks(S)$,$t_i \in Tasks(S)$. $\alpha_i \notin itrQ(S) \Rightarrow \alpha_i \in t_i$ $\alpha_j \notin itrQ(S) \Rightarrow \alpha_j \in t_j$ and $t_j \neq t_i$, or $\alpha_i \in itrQ(S)$, $\alpha_j \in itrQ(S)$ and $\alpha_i = \alpha_j$. Let $\sigma_i = C_0,\alpha_0,\cdots,\alpha_{i-1},C_i = (V_i, Q^{\cap} Q_\sigma, B_i, P_i)$. By Lemma 10 $GetPrefix(C'_0)$ returns a prefix p such that contains a trace $\sigma'_i = c' = c'_0,\alpha'_0,\cdots,\alpha'_i,\cdots,\alpha'_{n-1},c'_n$, $c'_n = (V_i, Q'^{\cap} Q'_\sigma, B_i, P_i)$ where $\sigma_i \equiv_{st_\varphi} \sigma'$ and $Q_\sigma \simeq_{st} Q'_\sigma$. Then considering $\sigma_{ii} = C_i,\alpha_{i+1},\cdots,\alpha_{n-1},C_n = (V_\sigma, Q_\sigma, B_\sigma, P_\sigma)$ σ_{ii} has n tasks and by I.H. and by transitivity of stuttering there exist a $\sigma'_{ii} = C'_i,\alpha'_{i+1},\cdots,\alpha'_{n-1},C'_n = (V_\sigma, Q'_\sigma, B_\sigma, P_\sigma)$ such that $\sigma_{ii} \equiv_{st_\varphi} \sigma'_{ii}$ and $Q_\sigma \simeq_{st} Q'_\sigma$ □

Theorem 4. *Let $\mathcal{T}$ be the transition system of $\mathcal{N} = \langle S_0, \cdots, S_N$. 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.*

Proof Let ψ be the set of propositions belonging to φ we will proof that for any trace σ from $\mathcal{T}$ $\sigma = C_0, \alpha_0, \cdots, \alpha_n, C_{n+1}$ there exist a trace $\sigma' = C'_0, \alpha'_0, \cdots, \alpha'_n, C'_{n+1}$ such that $\sigma \equiv_{st_\varphi} \sigma'$. We proof it by induction in the number of changes in the valuation of ψ

Base Case If the number of changes in the valuation of ψ in σ is equal to zero, it means that $L(C_0) = \cdots = L(C_n)$. Since C_0 belongs to $\mathcal{T}'$ then let $\sigma' = C_0$ and $\sigma' \equiv_{st_\varphi} \sigma$.

Induction Step Suppose that the number of changes in the valuation of ψ in σ is m then there exits a $\sigma' = C_0, \alpha'_0 \cdots \alpha'_n, C_{n+1}$ such that $\sigma \equiv_{st_\varphi} \sigma'$ then it also holds for $m+1$ changes in the valuation of ψ in σ.

Let $\alpha_i \in \Sigma S_i$ the first action in σ such that $\alpha_i \notin safe(\varphi)$. Let $\alpha_{k_0}, \cdots, \alpha_{k_m}$, for all k_j, l, $0 \leq k_j \leq i$, $0 \leq l \leq N$ $\alpha_{k_j} \in S_l$ the last extendable action from a task $t_p \in Tasks(S_l)$, $t_p \not\subset_{GI} S_l$ (1) where each k_j is ordered as follows: given two last extendable actions $\alpha_a \in S_{\alpha_a}$, $\alpha_b \in S_{\alpha_b}$ $S_{\alpha_a} \neq S_{\alpha_b}$, let suppose α_a is a *Send* action and $\alpha_b \in t$, $t \not\subset_{RI} \mathcal{S}_{\alpha_b}$ then if there exist α'_b which is a receive interrupt handler in S_{α_a} and $a < b' < b$ then $k_b < k_a$ otherwise k_a, k_b. The same reasoning is applied when α_b is *Send* action and $\alpha_a \in t$, $t \not\subset_{RI} \mathcal{S}_{\alpha_a}$. If α_a and α_b are both *Send* actions or they belong to a task $t \not\subset_{RI} \mathcal{S}_{\alpha_b}$ 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 s $0 \leq s \leq k_0$ and the trace $\cdots$, C_{s-1}, α_{s-1}, C_s,α_s, C_{s+1},$\cdots$ is equivalent to the trace $\cdots$,C'_{s-1},α_s,C'_s,α_{s-1},c'_{s+1},$\cdots$ and it is possible to get a trace $\sigma_{k_0} = c_0, \alpha'_0, \cdots, \alpha_{k_0}, C'_{k_0}$ such that for $0 \leq j \leq k_0$, $\alpha_j \in \Sigma_{S_l}$. Let $CV = \langle p_0, \cdots p_l, \cdots, p_N \rangle = GetNewCV(c_0)$ by Algorithm 2, Theorem 3 and Lemma 10 there exits a traversal of p_l $\sigma'_{k_0} = C_0, \alpha''_0, \cdots, C'''_{k_0}$ such that

App	Property	Size	Result	#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	$< 6 \times 10^{-4}$
	$\Box$(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 1. Experiment Results with NesC@PAT

#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

$\sigma_{k_0} \equiv_{st_\varphi} \sigma'$. Repeating this for all k_j in (1) and by transitivity of stuttering [18] we get that $\sigma_{k_n} = C_0, \alpha_0, \cdots, \alpha_n, C_{n+1} \equiv_{st_\varphi} \sigma'_{k_n} = C_0, \alpha'_0, \cdots, \alpha_{k_n}, C_{k_n+1}$. Permuting again $\cdots, C_{s-1}, \alpha_{s-1}, C_s, \alpha_s, C_{s+1}, \cdots$, for all s $k_{n+1} \leq s \leq i$ and by Algorithm 2, Theorem 3, and Lemma 10 $\sigma_i = C_0, \alpha_0, \cdots, \alpha_i, C_{i+1} \equiv_{st_\varphi} \sigma'_i = C_0, \alpha'_0, \cdots, \alpha'_i, C'_{i+1}$ and the number of changes in the valuations of ψ is n. By I.H. and transitivity of stuttering $\sigma \equiv_{st_\varphi} \sigma'$. □

6 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. NesC@PAT, the experimental data and related documents can be obtained from [1].

6.1 Enhancing NesC@PAT with Two-level POR

In this subsection, we present the verification results of an anti-theft application and the Trickle algorithm [16]. The anti-theft application is taken from the TinyOS distribution, which is a real-world application of sensor networks. It consists of more than 3000 LOC of the NesC program running on each sensor. The Trickle algorithm is widely used for code propagation in SNs, and we adopted a simplified implementation to show the reduction effects. Experiments were conducted on a PC with Intel Core 2 Duo CPU (2.33GHz) and 3.25GB memory with Windows XP.

For the anti-theft application, we checked if a sensor turns on its theft led whenever a theft is detected, i.e., $\Box(\text{theft} \Rightarrow \Diamond\text{alert})$. In the Trickle algorithm, we checked that eventually all the nodes are updated with the latest data among the network, i.e., $\Diamond \mathit{AllUpdated}$. We also checked a safety property to guarantee that each node never performs a wrong update operation. These properties are verified using NesC@PAT with the two-level POR. Results are presented in Table 1. The results of the safety property against the Trickle algorithm are presented in Table 2 for the comparison with T-Check. In Table 1, column *OH* shows the computational overhead for static analysis at compile time, which depends on the program size, the 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 so that we can use it to calculate *POR ratio* ($=\frac{\#State\ wt\ POR}{\#State\ wo\ POR}$). For safety properties, $\#State\ wo\ POR$ is estimated as $S_1 \times S_2 \times \cdots \times S_n$, where S_i is the state space of the i^{th} sensor; whereas for LTL properties, it is further multiplied by the size of the Büchi automaton of the corresponding LTL property. Note that this estimation is an under approximation since the state space of a single sensor is calculated without networked communication. Therefore, the *PORRatio* (both in Table 1 and 2) is also an under approximation. With this under-estimation, our POR approach achieves a reduction of at least 10^2-10^6. Further, the larger a network is, the more reduction it will be.

6.2 Comparison with T-Check

We compared the performance of our POR approach and the POR method implemented in T-Check, by checking the Trickle algorithm for the same safety property, on the same testbed running Ubuntu 10.04. We focus on reachability analysis as T-Check lacks support of LTL. We approximated the POR ratio by the number of explored states, i.e., $\mathit{POR\ Ratio} \approx \frac{\#State\ wt\ POR}{\#State\ wo\ POR}$, because T-Check adopts stateless model checking. Moreover, there is no way to calculate the 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. 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 the 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.

7 Related Work

This work is related to tools/methods on exploring state space of sensor networks. 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] 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. verified the *ESAWN* protocol by producing abstract behavior models from TinyOS applications, and uses the C model checker CBMC to verify the models [23]. The original *ESAWN* consists of 21000 LOC, and the abstract behavior model contains 4400 LOC (including both C code and CBMC statements). Our approach is comparable to this approach, since we support SNs with thousands of LOC on each sensor to be explored. However, this approach is only dedicated for checking *ESAWN* protocol and it abstracts away all platform-related behaviors. Tos2CProver [4,5] translates embedded C code (which are generated by the NesC compiler from NesC source code) to standard C, which is then verified by CBMC. Partial order reduction (POR) is used to reduce state space, and POR's over-approximation is improved by reachablity checking. Our work differs from this work in that this work only supports single-node TinyOS applications instead of the whole network. T-Check [17] is built on TOSSIM [15] and check the execution of SNs by DFS or random walk to find a violation of user-specified properties. T-Check adopts stateless and bounded model checking and is efficient to find bugs, and it was used to verify several TinyOS applications and revealed some 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. Our approach

complements T-Check as we propose 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.

8 Conclusions

In this paper, we proposed a two-level POR to reduce the state space of SNs significantly, based on the independence relations of actions between different sensors and inside a single sensor. We extended cartesian semantics to deal with concurrent systems with multiple levels of nondeterminism such as SNs. POR was then developed based on static analysis of independence and the sensor network cartesian semantics, and 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 direction is to apply abstraction techniques like [20] to obtain an abstracted model before applying POR, and the other is to adopt BDD techniques to encode symbolic state space.

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.