SYSTEMATIC AND AUTOMATIC VERIFICATION OF SENSOR NETWORKS

MANCHUN ZHENG

NATIONAL UNIVERSITY OF SINGAPORE

2013

SYSTEMATIC AND AUTOMATIC VERIFICATION OF SENSOR NETWORKS

MANCHUN ZHENG
(*B.Sc., South China University of China of Technology and Design, 2008*)

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2013

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Manchun ZHENG
08 June 2013

To my parents and husband.

Acknowledgements

This thesis would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this thesis.

First and foremost , I would like to give thanks to God for accompanying me and strengthening me to overcome all difficulties throughout the Ph.D journey.

My deepest and heartfelt gratitude goes to my supervisor, Dr. Dong Jin Song for his stimulating guidance, continuous suggestions and constant encouragement. He has walked me through all the stages of my doctoral program, leading me into the world of formal methods and model checking. Without his immense support in various ways, I would not have completed the writing of this thesis.

I'm greatly indebted to my mentors Dr. Sun Jun and Dr. Liu Yang, who have instructed and helped me a lot in the past five years. I thank them for introducing me to the exciting area of sensor network verification. Their supervision and involvement in this work has triggered and nourished my intellectual maturity, which would be beneficial for the rest of my life. My sincere appreciation also goes to Dr. David Sanán for his involvement and crucial contribution. He has helped me a lot in the past years and contributed a lot to the progress of this research.

I would like to express my great gratitude to Dr. Chin Wei Ngan and Dr. Chan Mun Choon for their valuable suggestions and insightful comments on my research work. I have special thanks to Dr. Gu Yu, Dr. Chen Chunqing and Mr. Luu Anh Tuan for their research collaborations. I'm grateful to my senior Dr. Zhang Xian and fellow student Zhang Shaojie for their support and friendship throughout these years.

My sincere appreciation also goes to my beloved friends from churches and Christian fellowships. Their continuous prayers, support and care have helped me to get through all circumstances. Especially, I want to thank my spiritual mentors Ms. Josephine Siao and Dr. Esther Goh.

I'm deeply indebted to my parents, my sister and my brother for their encouragement, support and love. Last but not least, my heartfelt appreciation goes to my husband Lin Hai Ting, who has been a spiritual friend, lover and schoolmate that supports and loves me in everything during my PhD journey.

Contents

Summary

Sensor networks (SNs) are experiencing increasing application nowadays, many of which are performing critical tasks like fire detection, surveillance monitoring, etc. In this thesis, we investigate how software verification techniques could be adopted to systematically model check SNs implemented in NesC/TinyOS, which is one of the most widely used platforms for developing SNs.

Firstly, we translate a NesC program to a semantically equivalent specification in Stateful Timed CSP (STCSP), and hence model checkers for STCSP like PAT can be used to formally verify the NesC program. This work analyzes and formalizes the execution model of TinyOS applications and defines mapping rules between NesC and STCSP. However, this approach suffers from several drawbacks including overhead introduced in the translation, difficulty in mapping the verification results to the original NesC program, and so on.

In order to directly examine the behaviors of SNs thoroughly, we develop the direct verification approach so that the capability of revealing errors/bugs could be maximized. On one hand, the semantics of the NesC language is formalized and formal definitions are presented to describe the event-driven execution model of TinyOS applications. On the other hand, we propose a compositional method to build a sensor network model from individual sensors with a given topology. The complete semantics of SNs is defined as a set of 66 firing rules. This formalization is then used to obtain the state space of a given sensor network, with fine-grain behaviors such as updating a variable. Model checking algorithms are developed to verify SNs against safety properties and liveness properties.

In order to perform efficient verification, we propose a novel two-level partial order reduction (POR) approach for SNs, which reduces unnecessary interleaving among sensors and among interrupts and tasks within each individual sensor. We have proved that our POR approach is sound and complete, preserving LTL-X properties. This approach has provided the opportunity for developing model checking techniques directly on NesC programs through the formal semantics of SNs. Moreover, the two-level POR greatly reduces the state space of SNs, making verification tasks efficient and effective. This novel POR approach could be extended for systems with various levels of concurrency.

Our work has been implemented as the domain-specific model checker NesC@PAT. We have studied the performance of NesC@PAT by verifying a number of real-world applications, and quantitatively compared the reduction ratio of our tool with a similar tool T-Check [89]. The results have shown that the POR approach is able to achieve reduction by a factor of at least 10^2-10^{10} and that our POR approach outperforms T-Check. The two-level POR approach significantly improves the performance of NesC@PAT, allowing SNs with programs of thousands of LOC (lines of code) to be fully verified. We believe that NesC@PAT can be used by SN developers to conveniently verify their SN programs before network deployment and thus will help improve the reliability of SNs.

Key words: **Sensor Networks, Formal Verification, Software Verification, Model Checking, Partial Order Reduction, Cartesian Semantics, TinyOS, NesC, Sensor Network Protocol**

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Sensor networks (SNs) are built on small sensing devices (i.e., sensors), which are then distributed in outdoor or indoor environments to conduct certain tasks. Recently, SNs have been increasingly used to develop various safety-critical systems, such as railway signaling, enemy intrusion detection, autopilot, fire detection, landslide detection and so on [6]. Such systems are usually expected to run unattended for a long period like several months or even years. Henceforth, failures or errors of these systems might cause severe damage to the environment and even human lives. Thus, it is significantly important to develop reliable and correct SNs, which, however, is highly challenging in the following aspects.

First, the behavior of a sensor is usually interrupt-driven. For example, a sleeping sensor might be waken up by a packet arrival. Most interrupts are related to uncertain external events like packet arrival, data sampling and so on. TinyOS [87], one of the most widely used operating systems for sensor networks, provides an interrupt-driven execution model for SN applications. NesC [53], the programming language of TinyOS applications, provides fine-grained control over the underlying devices and resources. The interrupt-driven feature has made the behavior of a single sensor complex and unpredictable.

Second, sensors are highly distributed in SNs and are allowed to independently execute concurrently. This loose concurrency among sensors make the behavior of an SN complicated and difficult to be analyzed thoroughly.

Third, sensor networks are usually running in unreliable environments and it is difficult to analyze SNs without sufficient information of the environments.

A number of simulating and debugging tools have been proposed for analyzing TinyOS applications, such as TOSSIM [86], TOSSF [128], etc. However, such tools lack the ability for finding bugs/errors thoroughly, especially those occurring at rare and unexpected scenarios like an overflowing buffer. Considering the fragment of a NesC program shown in Figure 1.1(a) where the statement *post sendTask*() tries to enqueue the task *sendTask* to TinyOS's task queue, if the *post* fails, then the statement *post sendTask*() will never have the chance to execute again, because the variable *sendTaskBusy* remains *True*. The most probable reason of the failure of a *post* statement is the overflow of the task queue, which

```
1  result_t tryNextSend(){
2      atomic{
3          if(!sendTaskBusy){
4              post sendTask();
5              sendTaskBusy = TRUE;
6          }
7      }...
8  }
9  void task sendTask(){
10     ...
11     sendTaskBusy = FALSE;
12     ...
13 }
```

(a) Buggy code

```
1  result_t tryNextSend(){
2      atomic{
3          if(!sendTaskBusy){
4              if(SUCCESS != post sendTask())
5                  sendTaskBusy = FALSE;
6                  else sendTaskBusy = TRUE;
7              }...
8  }
9  void task sendTask(){
10     ...
11     sendTaskBusy = FALSE;
12     ...
13 }
```

(b) Revised code

Figure 1.1: A Motivating Scenario

is extremely rare though could lead to severe consequences. Therefore, it is significant to be aware of the existence of this scenario. However, it is very difficult to identify such a specific issue by merely testing or simulating. Thus, a more powerful approach, other than testing or simulating, is required, so as to explore *all* possible scenarios of a sensor network to detect errors caused by rare but critical events.

Due to the highly distributed nature of SNs, it is difficult and complex to supervise or analyze their behaviors at run time. Therefore, apart from bug detection, it is significant for developers to know whether SNs are faithful to their requirements and system specifications before deployment. For example, in a fire detection system, it is desirable that *whenever* the system "feels" that there is fire then the alarm should be ringing. Another example is in a railway signaling system, it is crucial that *whenever* a train enters an available section, a blocked signal of that section should be released. Such problems are referred to as *temporal* properties because they are qualified in terms of time, described using terms such as *whenever*, *eventually* and so on. To solve such problems, one needs to thoroughly examine the behavior of SNs. Traditional approaches like testing, debugging or simulating only check limited scenarios of SNs and thus are incapable of solving this problem. Again, there is a need for approaches that systematically and exhaustively examine the system space of an SN and answer the question if the SN satisfies specific temporal properties.

Software verification techniques [11] have been proposed and adopted to successfully verify many systems and projects (e.g., [72]). Model checking [11] is one of the most widely used verification techniques, which checks desirable properties by systematically exploring the complete state space of the given system. On one hand, model checking systems against safety properties will reveal all possible safety violations, i.e., bugs or errors. One recent success is the full verification of the Intel i7 chip using model checking techniques [78]. On the other hand, model checking systems against liveness properties will answer the question if some desirable behavior will eventually happen. Liveness properties are usually

expressed in temporal logic, like Linear Temporal Logic (LTL) [105] and Computation Tree Logic (CTL) [31]. For example, an LTL formula "$\Box\Diamond send \wedge \Box\Diamond receive$ " will check if a sensor is able to send and receive new messages infinitely often. If one wants to find the potential bug mentioned in Figure 1.1(a), a temporal property saying that *sendTaskBusy* should be set to False *infinitely often* can be defined, i.e., "$\Box\Diamond(sendTaskBusy = False)$". Model checking techniques will then explore the whole system state space to decide whether a counterexample could be evidenced. With the counterexample, one may resolve the bug with the revised code shown in Figure 1.1(b).

The ability for finding errors of verification techniques has made formal verification more and more popular in many domains, including sensor networks. A number of approaches and tools have been published for modeling and verifying SN applications or SNs. And they could be grouped into three categories.

The first is to manually establish formal models for SN applications using various formal specification techniques [120, 133, 13, 106, 63, 122, 121], and then use an off-the-shelf model checker to perform verification. Such approaches contribute to the analysis of sensor network systems, but they are limited in two aspects. First, these approaches rely on manual modeling of SNs which is non-trivial and could cause false results. Second, since the formal relationships between formal models and SN programs are absent, these approaches have no guarantee of the correctness of the final implementation even if the model has been refined and verified to be completely correct.

Approaches in the second category model, analyze and verify SNs with NesC programs specifically for TinyOS. TinyOS applications have been modeled as hybrid automata [38], interface automata [158, 159], CSP [107], LOTOS [132], etc. Such approaches have helped developers to model and analyze sensor network systems, but they still cannot guarantee the correctness of the implementation code that is put into practice, because the formal relationships between formal models and source code are missing. Further, most of these approaches are not implemented as automatic tools and thus they require extra human effort for building models.

The approaches of the third category are automatic analysis or verification tools for sensor networks. There have been approaches for interface contract [8], finite state machine generation [80], security protocol implementation verification [66, 69], source code verification [111, 162, 26, 89, 27] for SNs, etc. Although these existing approaches have contributed a lot to analyzing and finding bugs of TinyOS applications or SNs, few of them simulate or model the interrupt-driven execution model of TinyOS. Further, only a few are dealing with the whole network rather than individual sensors, and the former is obviously more complex.

Model checking is often applied to high-level modeling languages like CSP [74], LOTOS [20], Promela [75] and so on, that describe the behavior of the system under analysis. Manual translation of implementation code into formal models is a non-trivial task and can

introduce additional errors that do not exist in the original code. Therefore, automatic and direct verification of implementation code is important and necessary. Consequently, it would be beneficial to develop an automatic verification tool for TinyOS applications of SNs. However, the complex system behavior of SNs usually causes the infamous state space explosion problem, and thus makes verification highly challenging.

The state space of a sensor network can be very huge. For example, given a sensor network consisting of n sensors, each of which has m states, the size of the state space is in the order of n^m. 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 states, 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 [25, 27] is limited to a single sensor, whereas the approaches in [69, 107, 89, 172] work only for small networks. The solutions in [111, 162] rely on aggressive abstraction techniques and thus is limited to particular properties only. Furthermore, T-Check [89] which is based on stateless model checking that keeps no track of explored states takes days or even months to detect a faulty state.

In an SN, the only shared or "global" resource among sensors is the message buffer of each sensor, because one sensor sending a packet may update the message buffer of another sensor. This nature makes partial order reduction techniques nicely suitable for SNs. For example, the interleaving among sensors could be neglected if there is no communication involved [89]. However, existing partial order reduction approaches for SNs only employ a small portion of possible reduction [24, 89], and thus more rigorous reduction techniques are in need.

In this work, we study how to make use of the powerful verification capability of model checking techniques for developing SNs with high reliability and correctness, tackling the verification challenge through appropriate reduction techniques.

1.1 Objectives

There have been a number of approaches for modeling and verifying sensor networks. Although some of the existing approaches were able to detect and reveal bugs of some TinyOS applications [89], there are still research gaps in verifying SNs, summarized as follows.

- Few of the current approaches simulate or model the interrupt-driven execution model of TinyOS, and thus are incapable of checking bugs/errors introduced by the concurrency of interrupts.
- Only a few [69, 111, 162, 89] are dealing with the whole networks, which are obviously more complex than individual sensors, since networked behaviors, including the concurrent execution and the communication among sensors, have made analysis more complex and challenging.

- The state space of sensor networks is incompletely explored and existing approaches are limited to debugging rather than verification. However, the complete exploration of the state space is non-trivial and highly challenging due to the state space explosion problem.
- There is no support for checking temporal properties, such as "*whenever* a packet is sent, it should *eventually* received by some sensor". Checking temporal properties is more complicated compared to non-temporal ones.
- The rules for conducting state space reduction of existing approaches are naive, e.g., only identifying unnecessary interleaving among sensors based on communication patterns [89]. The performance of reduction techniques could be further improved by handling intra-sensor concurrency and there is a need for an enhanced reduction approach.

Goals To summarize, among existing approaches no one verifies SNs completely and efficiently. Consequently, the main goal of this research is to develop a systematic and self-contained approach for completely and efficiently verifying SNs. In other words, we want to exhaustively analyze NesC programs in order to detect bugs/errors thoroughly, in an efficient way with satisfactory performance and verification speed. However, it is non-trivial and highly challenging to achieve this objective, due to the following reasons:

- The syntax and the semantics of the NesC language are complex [53] compared to those of formal modeling languages. To the best of our knowledge, there has not been any formal semantics for the NesC language. Thus establishing formal models from NesC programs is non-trivial.
- TinyOS provides hardware operations on motes (i.e., sensors) which can be invoked by NesC programs including messaging, sensing and so on [54, 52]. Therefore, modeling NesC programs (executing on TinyOS) requires modeling the behaviors of hardware at the same time.
- TinyOS adopts an interrupt-driven execution model, which introduces local concurrency (i.e., intra-sensor concurrency) between tasks and interrupts, increasing the complexity of model checking NesC programs.
- Inter-sensor concurrency and intra-sensor concurrency have made the system state space increase exponentially, which leads to the state space explosion problem easily.

Scope In this thesis, we will only focus on the approaches for verifying SNs in TinyOS/-NesC, not considering the verification of SNs in other platforms like Contiki [111]. This is because TinyOS is one of the most widely used platforms for developing SNs, and the

methodologies studied in this thesis could be applied to the verification of SNs on other platforms. Moreover, the features of SNs not considered in our work include energy analysis, hardware errors, node failure or reboot, node movement and so on, for the reasons that our work focuses on the analysis of SN implementation with the assumption that hardware devices are working well, and that we focus on non-probabilistic verification approaches which are incapable of analyzing failure behavior.

Our work contains a translation-based approach for modeling and verifying TinyOS applications, which translates NesC programs into STCSP (Stateful Timed CSP) [143] models and uses the model checker PAT [144] to perform verification tasks. However, this approach only supports single-node applications and the scalability is highly limited because of the redundant states introduced by the semantic difference. Thus there is a need for direct-verification approaches.

To perform direct verification for sensor networks, we formalize the semantics of sensor networks. The formalization of SNs includes a component model library for hardware, and the formal definitions of NesC programs and the TinyOS execution model. Based on these, the labeled transition systems (LTSs) of individual sensors are constructed directly from NesC programs. With a network topology that specifies how the sensors are connected, the LTS of an SN is then composed (on-the-fly) from the LTSs of individual sensors. The complete formal semantics of SNs is defined as a set of 66 firing rules, including rules for NesC language constructs, the TinyOS execution model, device concurrency and network composition. Model checking algorithms are adopted to support the verification of safety and liveness properties, specified as state reachability and LTL [105] formulas. Both the state space of an individual sensor and that of the whole SN can be explored for verification.

In order to support the analysis of large programs and larger networks, we propose a novel two-level partial order reduction (POR) approach which takes advantage of the unique features of SNs as well as NesC/TinyOS. Existing POR methods [61, 34, 50, 164, 65] 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 order, producing numerous states. Applying POR for interrupts is highly nontrivial because all interrupts would modify the task queue and lead to different sequences 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). We also show that the two-level POR is sound and complete for LTL-X properties, i.e., properties specified as LTL formulas without the next operator.

Our approach has been implemented as the NesC module in PAT [97, 144, 99], named NesC@PAT. The core of NesC@PAT consists of 153K lines of C# code, exclusive of its GUI implementation. We study the performance of NesC@PAT by a number of real-world applications such as the Trickle algorithm [88], an anti-theft application, etc. We also quantitatively compare the reduction ratio of our tool with a similar tool T-Check [89]. The results show that our POR approach is able to achieve reduction by a factor of at least 10^2-10^{10} and that our POR approach outperforms T-Check. The two-level POR approach significantly improves the performance of NesC@PAT, allowing SNs with programs of thousands of LOC (lines of code) to be fully verified.

1.2 Contributions

The results of this thesis would be beneficial to SN developers by providing them with an automatic and complete tool to verify SN implementation against various properties. Moreover, the semantics of SN implementations formalized in this research could contribute to bridging low-level programming languages and formal modeling approaches. Furthermore, the two-level POR approach presented in this thesis could be extended and applied to other concurrent systems, making software verification feasible for complex systems. We highlight our contributions as follows.

- The translation-based approach defines a set of mapping rules from NesC language to STCSP models, which provides the opportunity to analyze or verify sensor network applications with existing model checkers. And the translation methodology could be reused by other researchers to translate NesC programs into formal models other than STCSP.
- We formally define the operational semantics of NesC, TinyOS and SNs. New semantic structures are introduced for modeling the TinyOS execution model and hardware-related behaviors like timing, messaging, etc.
- The direct-verification approach works directly on NesC programs, without building abstract or formal models before applying verification techniques. Manual construction of models is avoided, which makes our approach useful in practice.
- The interrupt-driven feature of the TinyOS execution model is preserved in the LTS's generated in our approach. This allows concurrency errors between tasks and interrupts to be detected.
- Our approach supports the verification of both safety and liveness properties. This provides flexibility for verifying different properties to guarantee the correctness of SNs. Moreover, the expressive power of LTL has allowed to define significant liveness properties (e.g., the infinite occurrences of an event).

- The two-level POR approach greatly reduces the state space of SNs, making verification tasks efficient and effective. This novel POR approach could be extended for systems with various levels of concurrency to achieve substantial reduction.

Publications

Most of the work presented in this thesis has been published in international conference proceedings or journals.

The translation-based verification of sensor network of Chapter 4 was presented in the 4^{th} IEEE International Conference on Secure Software Integration and Reliability Improvement SSIRI'10 (Jun 2010) [170]. The work in Chapter 5 was presented in the 13^{rd} International Conference on Formal Engineering Methods ICFEM'11 (Oct 2011) [172]. The tool NesC@PAT presented in Chapter 7 was demonstrated in the 9^{th} ACM Conference on Embedded Networked Sensor Systems SenSys'11 (Nov 2011) [173]. The work in Chapter 6 was presented in the 14^{th} International Conference on Verification, Model Checking, and Abstract Interpretation VMCAI'13 (Jan 2013) [171]. I also made contributions to other publications [155, 29, 30] which are remotely related to this thesis.

For all the publications mentioned above, I have contributed substantially in both theory development and tool implementation.

1.3 Thesis Structure

The rest of this thesis is organized as follows.

Chapter 2 provides the preliminary knowledge of this work. First, we introduce the important features and concepts of TinyOS and NesC, explaining how the NesC language supports TinyOS's interrupt-driven execution model. Second, software verification techniques, especially model checking techniques, are briefly introduced, as well as the assertion annotation language that is used in our work to define verification goals.

Chapter 3 provides the related work review, including approaches for modeling and analyzing sensor networks, NesC programs, TinyOS applications and partial order reduction techniques for various systems and programming languages.

Chapter 4 presents the translation-based verification approach of SNs, which automatically obtains STCSP models from NesC programs and uses the model checker PAT to verify the generated models.

Chapter 5 discusses the direct verification approach for SNs, which obtains LTS's from sensor networks running NesC programs. First, the operational semantics of NesC language constructs is defined in the form of firing rules. In addition, specific operators are introduced to obtain the behaviors of TinyOS's interrupt-driven execution model. Second, the network compositional rules are developed to describe networked behaviors. After that, the verification algorithms for safety and liveness properties of PAT are reused to perform

verification tasks. We also study how this approach verifies sensor networks implemented with a code propagation protocol, i.e., Trickle. Experiments are carried out with different network topologies and network sizes, and results show that this approach is able to detect previously unknown bugs.

Chapter 6 discusses a two-level POR approach for optimizing the performance of verification. Networked actions and intra-sensor actions are analyzed to identify their commutativity, so that the state space of a given sensor network can be reduced. We refer to the analysis of commutativity as independence analysis in our work. Based on the independence analysis, we propose the sensor network cartesian semantics which obtains a state space that is smaller but sound for a certain property. We study the performance of the POR method, and a comparison with the similar tool T-Check [89] is also presented.

Chapter 7 presents the domain-specific model checker NesC@PAT, which verifies sensor networks against safety properties and liveness properties. The two-level POR approach presented in Chapter 6 is integrated in NesC@PAT to obtain better performance. The translation-based approach described in Chapter 4 is included in NesC@PAT as well, providing the comparison of direct verification and translation-based verification.

Chapter 8 concludes the thesis with recommendations for applications and future research directions.

Chapter 2

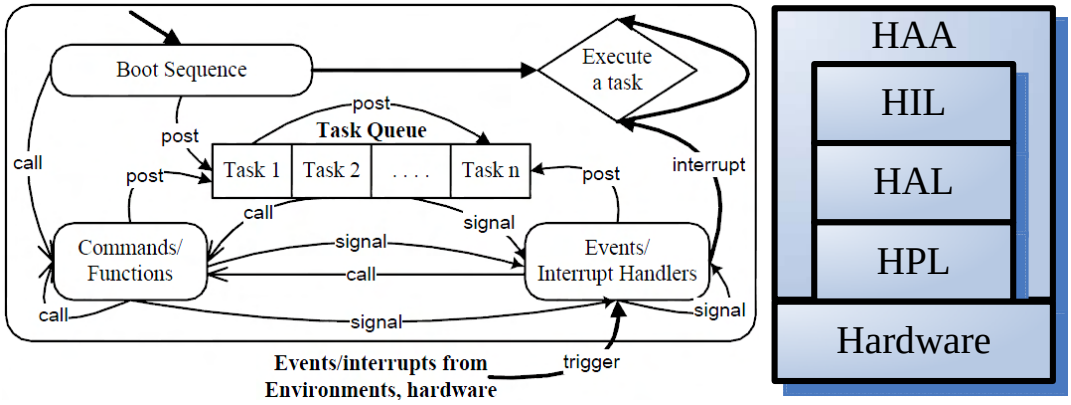
Preliminaries

This chapter contains the preliminary knowledge of this thesis. Sections 2.1 and 2.2 introduce TinyOS and NesC, the operating system and programming language for sensor networks, respectively; Section 2.3 discusses how interrupts are implemented by TinyOS/NesC and how we abstract interrupt handlers for modeling SN applications; and Section 2.4 introduces general software verification and model checking techniques.

2.1 TinyOS

TinyOS [40, 87, 54, 52, 85] is an operating system for embedded systems, which was designed to make the best use of limited resources of hardware, and to support platform-independent programming by dealing with low-level operations of different sensor platforms including Micaz, Iris, Teslob, etc. The platform-independency of TinyOS has attracted many SN developers and now there are a large number of sensor network applications implemented based on TinyOS, such as the Collection Tree Protocol [57], the Trickle algorithm [88], and so on. TinyOS implements an interrupt-driven execution model for SN applications, with the concepts of tasks, commands and events, as illustrated in Figure 2.1(a). Tasks are deferred computations, which are scheduled in a certain order, e.g., FIFO (first in first out). Tasks always run until completion, i.e. only after a task is completed, can a new task start its execution. In contrast, interrupts are preemptive and always preempt tasks. Commands and events are similar to traditional functions, except that they are related to interfaces and components. We will discuss the concepts of commands and events in Section 2.2 where we introduce the NesC language.

TinyOS is implemented in NesC, with a component library for hardware operations like sensing, messaging, timing, etc. The TinyOS component library adopts a three-layer Hardware Abstraction Architecture (HAA), including Hardware Presentation Layer (HPL), Hardware Adaptation Layer (HAL) and Hardware Interface Layer (HIL) [87, 52], as shown in Figure 2.1(b). The design of HAA gradually adapts the capabilities of the underlying hardware platforms to platform-independent interfaces between the operating system and the

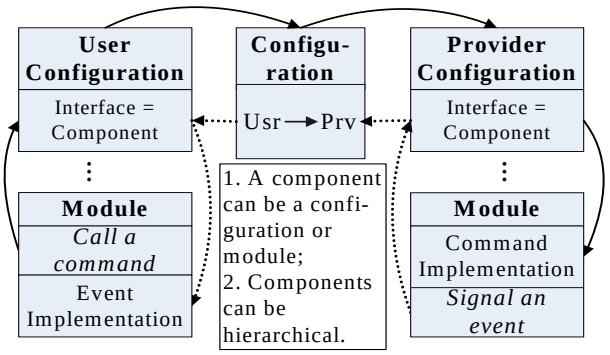


(a) Execution Model (b) Hardware Architecture

Figure 2.1: TinyOS

```
1  int main()...{
2      atomic {
3          platform_bootstrap();
4          call Scheduler.init();
5          call PlatformInit.init();
6          ...
7      }
8      // enable interrupts
9      nesc_enable_interrupt();
10     // execute boot event
11     signal Boot.booted();
12     // start the task scheduler
13     call Scheduler.taskLoop();
14     // terminates
15     return -1;
16 }
```

(a) TinyOS Boot Function



(b) Call graph of NesC programs

Figure 2.2: TinyOS and NesC

application code. To model TinyOS applications, it is unavoidable to model the behaviors of hardware operations in a way that is faithful to the component library.

Since version 2.0, each TinyOS application should contain a component *MainC* (predefined by TinyOS) that implements the boot function of a sensor [85]. Figure 2.2(a) sketches the function that implements the boot function. At first, the scheduler, the hardware platform and other necessary software components are initialized (lines 2-7). Then interrupts are enabled (line 9) and the event *booted* of interface *Boot* (*Boot.booted*) is signaled (line 11), after which the scheduler repeatedly runs tasks that have been enqueued (line 13). The execution of lines 2 to 7 is usually short and always handled by TinyOS. Therefore, our approach assumes that this part is always correct and the starting point of modeling the behaviors of a sensor is considered as the execution of the event *Boot.booted*.

NesC Construct	Example	Remark
Command	*command* error_t AMControl.start() {···}	There are commands, events and tasks besides ordinary functions. The only difference among them is the way of invocation. A task is a parameterless function without any return value.
Event	*event* message_t* Receive.receive (message_t* msg, ···, uint8_t len) {··· return msg; }	
Task	*task* void setLeds() {···}	
Call	*call* Timer.startPeriodic(250);	Call, signal and post are function calls, invoking commands, events and tasks, respectively.
Signal	*signal* Timer.fired();	
Post a task	*post* setLeds();	
Atomic	*atomic*{x = x + 1; call AMSend.send(dst, pkt);}	Interrupts are disabled within an atomic block.

Table 2.1: Common-used NesC Constructs

2.2 The NesC Language

NesC (Network-Embedded-System C) [53] was proposed to develop TinyOS applications, by embodying the structural concepts and the execution model of TinyOS. The interrupt-driven feature of TinyOS is achieved by tasks and interrupts in NesC. Besides, as a dialect of C, NesC inherits many C features like control statements, variable types, pointers and so on. These have made NesC source code complex and difficult to understand, and even tedious to analyze its correctness. The syntax of the NesC language can be found in Appendix A.

NesC has two basic modular concepts: interface and component. An *interface* declares a number of commands and events. A *command* denotes the request for an operation, and a corresponding *event* indicates the completion of that operation. Commands and events together are the basis of *split-phase* operations [53], which separate requests and responses for long services and operations. In this way, NesC achieves a non-blocking programming convention.

An interface can be either *provided* or *used* by a component. In NesC, there are two types of *component*, i.e. configuration and module. A *configuration* indicates how components are wired to one another via interfaces. A *module* implements the *commands* declared by its provided interfaces and the *events* declared by its used interfaces. Commands and events are two types of functions, and *task* is the third. A component may call a command or signal an event of an interface. Table 2.1 exemplifies the common-used constructs of the NesC language and more syntax of NesC can be found in [55].

A call graph describes the wiring relation between components. Figure 2.2(b) illustrates a general call graph of NesC programs. Inside a configuration, a second-level configuration can be wired to a third component, where the second-level configuration itself contains a wiring relation between a set of components. Thus, the call graph of a NesC program might

be a hierarchical "tree" of components, where intermediate nodes are configurations and leaves are modules.

Although NesC is an extension of the C language, it does not support advanced C features like dynamic memory allocation, function pointers, multi-threading and so on, which makes it an 'easier' target for formal verification. Nonetheless, it supports almost the same set of operators, data types, statements and structures as C does and, in addition, NesC-specific features such as calling a command, signaling an event, posting a task and so forth. Therefore, verifying NesC programs is highly non-trivial and the challenges are illustrated in the following.

- Function calls like calling a command or signaling an event can be complex if the module invoking the command/event and that implementing it are wired via a hierarchical call graph.
- NesC allows local variables declared in functions or even in statement blocks, just like C does. Typically, compilers maintain a stack to handle variables of different scopes. However, it is tedious and complex to model the variable stack. A common solution is to carefully rename variables to avoid naming conflicts. This increases the complexity of the modeling procedure.
- NesC is a typed programming language, and all data types of C including array and struct are supported. There are also type operations (e.g. type casting) supported by NesC. Therefore, modeling NesC should take into account type operations.
- There are other expressive features of NesC, which are inherited from C that make it complex. Examples of such features include pointers, generic interfaces and components using types as parameters, definition of types, pre-compilation, etc [55].

We remark that our approach targets NesC programs and does not necessarily support the verification of C programs. In the following, we briefly explain how we handle the above "troubling" features.

Fortunately, NesC is static [53], i.e. no dynamic memory allocation or function pointers. Thus the variable access and the call graph can be completely captured at compile time. In our work, we treat a pointer as a normal variable, the value of which is a reference to a certain variable. We develop a parser to produce the call graph of a NesC program with the function (command, event, task or normal function) bodies defined by each component. A nested search algorithm is designed to traverse the call graph to obtain the corresponding function body once a function is invoked.

Local variables are modeled statically in our approach, with a renaming method to avoid naming conflicts. Nested function calls are supported, e.g., $foo(f(1), f(2))$ in Figure 2.3(a), with the assumption that there is no "cycle" within the calling stack. As for the function *foo* defined in Figure 2.3(b), our approach is unable to distinguish which function *foo* in the

```
1 void main(){
2     foo(f(1), f(2));
3 }
4 int foo(int a, int b){
5     return (a + b);
6 }
7 int f(int u){
8     return (u * 2);
9 }
```

(a) Nested Function Call

```
1 int foo(int u){
2     int v = u;
3
4     if(v < 100){
5         return v + foo(v+1);
6     }
7
8     return v;
9 }
```

(b) Recursive Function

Figure 2.3: Examples of Invoking Nested Function

calling stack that the variable v belongs to, because we rename a local variable according to the position of its declaration. Thus distinguishing the variable v in the recursive function *foo* in Figure 2.3(b) becomes costly and we get rid of such cases by restricting our approach to supporting code without this feature. Such restriction is modest, because the most common invocation cycle of NesC programs lies in the split-phase operations, i.e. when a command finishes it signals an event and in the end of the event the command is called again. However, NesC programmers are recommended to avoid such situations [85]. Even in such cases, we can still get rid of naming conflicts of local variables because a repeated invocation of a function is always at the end of the previous one.

Typing information is captured and we distinguish variables declared as different types and analyze functions with parameters being types. Our work also supports defining new types by *struct* and *typedef*. Moreover, pre-compilation is supported, as well as capturing information from *.h* files.

Example 1 (Trickle)**.** *Trickle [88] is a code propagation algorithm which is intended to reduce network traffic. Trickle controls the message sending rate so that each sensor hears a small but large enough number of packets to stay up-to-date. In the following, the notion code denotes the large-size data (e.g. a route table) each sensor maintains, while code summary denotes the small-size data that summarizes the code (e.g. the latest updating time of the route table). Each sensor periodically broadcasts its code summary, and*

- *lasts quiet if it receives an identical code summary;*
- *broadcasts its code if it receives an older summary;*
- *broadcasts its code summary if it receives a newer summary.*

We have implemented this algorithm in a NesC program *TrickleAppC*, with the modification that a sensor only broadcasts the summary initially, instead of periodically. A struct *MetaMsg* is defined to encode a packet with a summary, and *ProMsg* is defined to encode a packet with a summary and the corresponding code. Initially, each node broadcasts its

```
1  Module DeviceC{
2      provides interface Service;
3  }implementation{
4      command void Service.DoSth(){
5          //generate a hardware request
6      }
7      //upon hardware intterrupt,
8      //this task will be posted
9      task void Service_Done(){
10         signal Service.Done();
11     }
12 }
```

(a) Device Code

```
1  Module AppC{
2      uses interface Service;
3      uses interface Boot;
4  }implementation{
5      event void Boot.booted(){
6          call Service.DoSth();
7      }
8      event void Service.Done(){
9          //service is done,
10         //do sth else now
11     }
12 }
```

(b) Application Code

Figure 2.4: Pseudo Code of Hardware Service Implementation

summary (a *MetaMsg* packet) to the network. If an incoming *MetaMsg* packet has a newer summary, the sensor will broadcast its summary; if the received summary is outdated, the sensor will broadcast its summary and code (a *ProMsg* packet). An incoming *ProMsg* packet with a newer summary will update the sensor's summary and code accordingly.

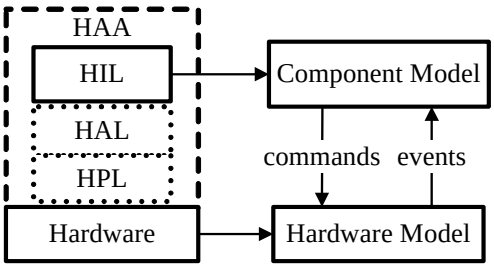
2.3 Interrupt Handlers

In NesC programs, there are two execution contexts, *interrupt handler* and *task* (a function), described as *asynchronous* and *synchronous*, respectively [53]. An interrupt handler can always preempt a task if interrupts are not disabled. In the TinyOS execution model [85], a task queue schedules tasks for execution in a certain order. In our work, we model the task scheduler in the FIFO (first in first out) order, since it is the default case of TinyOS and is widely used.

The code fragments shown in Figure 2.4 abstract the common mechanism that TinyOS implements hardware service. Usually, a device is implemented as a NesC component in the TinyOS component library, and we use the code shown in Figure 2.4(a) to sketch the implementation of a certain service, e.g, sending a packet, activing an alarm, etc. Due to the split-phase feature of NesC, an interface defined with commands and events will be provided to link the application code with the device component that implements a certain service. In Figure 2.4, the interface *Service* declares the command *DoSth* and the event *Done*. The *DeviceC* module provides *Service* by implementing the command *DoSth*, while the application component *AppC* uses the interface *Service* by implementing the event *Done*. These two components are wired via the interface *Service* in a configuration component which is not shown here. The 6^{th} line of *AppC* calls the command *DoSth* of the interface *Service*, which is the actual way how it "uses" *Service*. Since *AppC* is linked with *DeviceC* via the *Service* interface, calling *Service.DoSth* will execute the command body *Service.DoSth* implemented in *DeviceC*. During the execution of the command *DoSth*, the corresponding

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

(a) Abstract Interrupt Handler



(b) Hardware Service

Figure 2.5: Modeling Hardware Behavior

hardware request is sent to the hardware and the command terminates after that. Once the hardware request is completed, an interrupt will occur upon which the corresponding interrupt handler is executed.

Now we illustrate how interrupt handlers work in TinyOS. As shown in Figure 2.1(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 the application code. In our approach, we treat interrupt handlers as black boxes, as we assume that low-level 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 the pseudo code in Figure 2.5(a). For example in Figure 2.4(a), the device component implements the task *Severice_Done* to signal the event *Service.Done*. The interrupt handler caused by the related hardware service posts this task to the task queue, and later when the task *Severice_Done* is executed, the event body of *Service.Done* implemented by *AppC* will be executed and in this way the "control" will be given back to *AppC*.

In our work, the execution of an interrupt handler is considered as *one atomic 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. We also assume that 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).

To summarize, we model hardware at HAA's top layer, i.e. Hardware Interface Layer, ignoring differences between the underlying hardware platforms, as shown in Figure 2.5(b). A hardware interrupt is modeled as an atomic action which posts a task to the task queue. This task will signal the corresponding event for as a response to the interrupt. This is also the way that TinyOS deals with interrupts.

2.4 Software Verification

Although software testing and debugging have been widely used and improved for a long time, their ability to find bugs or errors is still limited, in that they only explore some (*not all*) possible scenarios. Software verification [11] is one of the most powerful techniques for checking the correctness of software (i.e., source code), because it automatically traverses all scenarios of the target system.

Model checking [11] is one of the most powerful approaches of software verification techniques, which explores the complete (or equivalently complete) state space of the target software system to find a violation of a given property like a deadlock. There are successful model checkers proposed to verify formal models instead of software in practice, such as SPIN [11], FDR [23], etc. Such approaches are able to handle large systems but yet unable to assure correctness of software at implementation level. More recently, model checkers for source code are being developed. Microsoft proposed SLAM for model checking C programs [12], while Havelund and his colleagues developed the model checker Java PathFinder for verifying Java programs [73].

In particular, model checking has been successfully applied to finding intricate errors of both software and hardware systems. For example, with the help of model checking techniques, NASA analyzed and found errors over several modules of a spacecraft control system [76, 72]. Another recent success is the full verification of the Intel i7 chip using model checking as the replacement of testing in the project [78].

In the following, we present the assertion annotation language defined in our work to represent verification goals.

2.4.1 Properties Specification

Our work supports the verification of both safety properties and liveness properties, in the form of state reachability and linear temporal logic (LTL) formulas. We propose a lightweight assertion annotation language for specifying these properties, which defines each property as an assertion.

The syntax of our assertion annotation language is presented in Figure 2.6. The keyword #define is used to define a logical formula with a certain name Var (line 3). Meanwhile, the keyword #assert is used to define an assertion. Lines 5 to 10 are the types of assertions supported by the language. First, line 5 defines properties in LTL formulas. Second, lines 6 to 10 are dedicated for state reachability properties, where lines 6 to 9 are to specify predefined state reachability properties and line 10 is to define user-specified state reachability properties based on the formulas defined by the keyword #define. The syntax of logical formulas, i.e., *mathExpr*, is presented in Appendix B.

```
1  assertions : verifyExpr+ ;
2
3  verifyExpr : #define var mathExpr | #assert assertionExpr ;
4
5  assertionExpr : var |= ltlExpr
6                | var never Terminates
7                | var never NullPointerAccess
8                | var never ArrayIndexOverflow
9                | var never InfiniteTask
10               | var never eventExpr
11
12 ltlExpr : (eventExpr | ltlOprEle)+ ;
13
14 ltlOprEle : ( | ) | [] | <> | && | || | -> | ! | <-> | \/ | /\ ;
15
16 eventExpr : eventNameExpr | eventEle ;
17
18 eventNameExpr : var (.var)* ;
19
20 eventEle : True | False | SUCCESS | FAIL
21          | var | (1 ... 9)(0 ... 9)* | [ | ] | . ;
22
23 var : (a .. z | A ... Z | _)(a ... z | A ... Z | _)* ;
```

Figure 2.6: Assertion Annotation Language

2.4.2 Propositional Logical Formula Definition

Propositional logical formulas are defined by the keyword *#define* in the following pattern:

$$\#define\ var\ formula,$$

where *formula* is a propositional logical formula and *var* is a reference if *formula*. Assertions will then referencing *formula* using *var*.

Variable Renaming Propositional logical formulas are composed of variables declared in the NesC programs. Since variables are declared within a component's scope, a renaming method for identifying variables from different components and different sensors is necessary. In our work, given a variable *var* declared in a component *comp* of the sensor $\mathcal{S}$, this variable *var* is renamed as $\mathcal{S}.comp.var$. As for default variables such as *TOS_NODE_ID* (the identifier of the sensor), *TOS_AM_ADDRESS* (the AM address of the sensor), they are sensor-wide variables and thus the *comp* field can be omitted in the renaming procedure. In this way, we are able to avoid confusion of variables declared by different components or sensors.

Example 2 (Definitions of formulas)**.** *In the Trickle application discussed in Example 1, the variable $\mathcal{S}_i.App.summary$ is initialized as $\mathcal{S}_i.TOS\_NODE\_ID$ and the larger value of summary denotes the summary of the newer version of code. We can define the following propositional formulas:*

1. *#define* $\mathcal{S}_1 SumChanged$ $\mathcal{S}_1.App.summary \neq \mathcal{S}_1.TOS\_NODE\_ID$;

2. $\#define \;\; \mathcal{S}_1 FalseUpdated \;\; \mathcal{S}_1.App.summary < \mathcal{S}_1.TOS\_NODE\_ID;$

3. $\#define \;\; FalseUpdated \;\; \mathcal{S}_1.App.summary < \mathcal{S}_1.TOS\_NODE\_ID$
   $|| \; \mathcal{S}_2.App.summary < \mathcal{S}_2.TOS\_NODE\_ID \; || \cdots || \; \mathcal{S}_n.App.summary < \mathcal{S}_n.TOS\_NODE\_ID;$

4. $\#define \;\; AllUpdated \;\; \mathcal{S}_1.App.summary = \mathcal{S}_n.TOS\_NODE\_ID$
   $|| \; \mathcal{S}_2.App.summary = \mathcal{S}_n.TOS\_NODE\_ID \; || \cdots || \; \mathcal{S}_n.App.summary = \mathcal{S}_n.TOS\_NODE\_ID.$

In the Trickle protocol, the variable *summary* should only be updated to a greater value, which is considered to be representing a newer version number. This variable *summary* is initialized with the value of TOS_NODE_ID of the current node.

The explanation of the formulas in Example 2 is illustrated as follows.

1. $\mathcal{S}_1 SumChanged$ holds if the variable *summary* of sensor $\mathcal{S}_1$ is changed after initialization;

2. $\mathcal{S}_1 FalseUpdated$ holds if *summary* of sensor $\mathcal{S}_1$ has a value smaller than $\mathcal{S}_1.TOS\_NODE\_ID$ which is the initial value of *summary* in sensor $\mathcal{S}_1$;

3. *FalseUpdated* holds if some sensor updates its *summary* with an older version number;

4. *AllUpdated* holds if the *summary* variables of all sensors are updated with the newest version number, i.e., $\mathcal{S}_n.TOS\_NODE\_ID$, because $\mathcal{S}_i.TOS\_NODE\_ID$ is assigned to be i in our example.

2.4.3 State Reachability

State reachability properties are dedicated to finding undesirable behaviors or wrong states of a sensor network. A state reachability property is specified by an assertion in the following form:

$$\#assert \;\; System \;\; never \;\; Goal,$$

where *#assert* and *never* are reserved keywords, and *Goal* is a reference to a propositional logical formula that has been previously defined. *System* is the reference of the target model to be checked, which could be a certain sensor or the whole network. The keyword *SensorNetwork* is reserved to represent the whole sensor network.

Example 3 (State Reachability Properties)**.** *Based on the formulas defined in Example 2, we can specify the following properties:*

- $\#assert \;\; \mathcal{S}_1 \;\; never \;\; \mathcal{S}_1 FalseUpdated;$
- $\#assert \;\; SensorNetwork \;\; never \;\; FalseUpdated.$

The first assertion checks if the sensor $\mathcal{S}_1$ might perform a wrong update on the variable $\mathcal{S}_1.App.summary$. This assertion is a single-sensor assertion, and during its verification, only the system state space of $\mathcal{S}_1$ is explored.

The second assertion is a network assertion, which requires the state space of the whole network to be examined during the verification phase. In this example, the assertion checks if the network system state space contains a state that satisfies the formula *FalseUpdated*.

Predefined State Reachability Property Similar to C programs, there are common problems with regards to memory accessing and system evolution in NesC programs, such as an exception caused by accessing a null pointer, an infinite loop making other processes/tasks hungry, etc. In order to allow such problems to be checked conveniently, we define a set of pre-defined properties to specify such problems, as shown in the following.

- Termination, specified by *#assert System never Terminates*, where *Terminates* is a reserved keyword. Generally sensors are expected to actively listen to the environment or passively stay sleeping until some external events are triggered. Therefore, if a sensor or a sensor network terminates, usually it means that the system has some bug.
- Array Index Overflow, specified by *#assert System never ArrayIndexOverflow*, where *ArrayIndexOverflow* is a keyword reserved for this property. This is to check if an array index overflow is encountered. Such cases should be avoided because during the execution of a NesC program, accessing an array by an overflow index will cause fatal exceptions.
- Null Pointer Accessing, specified by *#assert System never NullPointerAccess*, where *NullPointerAccess* is a keyword reserved for this property. This is to check if a null pointer is accessed. Similarly, accessing a null pointer will always trigger fatal exceptions.
- Infinite Tasks, specified by *#assert System never InfiniteTask*, where *InfiniteTask* is a reserved keyword. In TinyOS, tasks are run to completion and if there is a task containing an infinite loop, other tasks will have no chance to run. Therefore, we have to avoid such circumstances.

2.4.4 LTL Properties

Type	Operator	Name	Explanation
Unary	$X\varphi$	Next	φ should hold at the next state.
	$\Box\varphi$	Globally	φ should hold on the entire subsequent path.
	$\Diamond\varphi$	Finally	φ should hold eventually somewhere on the subsequent path.
Binary	$\varphi \cup \phi$	Until	φ should hold on the subsequent path at least until ϕ starts to hold.
	$\varphi R \phi$	Release	ϕ should hold until and at the point where φ first holds; if φ never holds then ϕ must remain true forever.

Table 2.2: LTL Operators [129]

Properties specified by LTL formulas are defined using the following form:

$$\#assert\ System\ \models\ LTLformula,$$

where $\models$ is a symbol dedicated for LTL checking, and the LTL formula is composed by LTL operators including $\Box$ (always, globally), $\Diamond$ (eventually, finally), X (next), $\cup$ (until) and R (release). The meanings of these LTL operators are given in Table 2.2.

Example 4 (LTL Properties)**.** *Based on the formulas defined in Example 2, we can specify properties in the following:*

1. $\#assert\ \ SensorNetwork\ \models\ \Diamond\ FalseUpdated$. *This assertion is valid iff the network eventually reaches a state that in which the formula FalseUpdated holds.*

2. $\#assert\ \ SensorNetwork\ \models\ \Box\Diamond\ FalseUpdated$. *This assertion is valid iff the network reaches a state in which the formula FalseUpdated holds for infinitely many times.*

Example 5 (An LTL Property)**.** *Now we define a formula $\mathcal{S}_1 CodeChanged$ in the following:*

$$\#define\ \ \mathcal{S}_1 CodeChanged\ \ \mathcal{S}_1.App.code \neq \mathcal{S}_1.TOS\_NODE\_ID.$$

This formula holds iff the value of variable code of sensor $\mathcal{S}_1$ is changed, since it is initialized as the value of TOS_NODE_ID. Based on this, we could further define the assertion:

$$\#assert\ \ SensorNetwork\ \models\ \Box(\mathcal{S}_1 SumChanged \Rightarrow \Diamond\, \mathcal{S}_1 CodeChanged).$$

This assertion is valid if and only if whenever the summary of $\mathcal{S}_1$ is changed, then eventually the code of $\mathcal{S}_1$ should be changed as well.

Chapter 3

Related Work

This chapter presents the related work of this thesis.

3.1 Formal Modeling and Analyzing SNs

This section provides the work that has been done for examining how formal techniques such as formal modeling and verification can be applied to sensor networks. According to the formalism used, this work is classified into four categories: process algebra, calculus, timed formalism, and probabilistic approaches, discussed in Section 3.1.1, Section 3.1.2, Section 3.1.3 and Section 3.1.4, respectively.

3.1.1 Process Algebra

Process algebras (or process calculi) [51] are a diverse family of related approaches for formally modelling concurrent systems. Process algebras provide the high-level description of interactions, communications, and synchronization between a number of independent processes. Algebraic laws have been proposed to allow process descriptions to be systematically manipulated and analyzed, so that formal reasoning about system properties could be performed. Leading examples of process algebras include CSP [74], CCS [109], ACP [19], LOTOS [132], etc. In the following, we discuss a collection of work in modeling and verifying (or analyzing) sensor network systems with process algebras including Promela [75], CSP and ReactiveML [104].

Oleshchuk demonstrates the applicability of the finite model-checking based approach for analyzing properties of ad-hoc sensor networks in [120]. A sensor network is considered as a network of communicating finite state machines and is presented as a set of concurrent communicating processes in Promela. Properties of ad-hoc sensor networks are expressed in the form of LTL formulae and are verified by SPIN [11]. This work has the emphasis on dynamic network changes and communication ability between sensor nodes but it neglects important features of sensor networks such as broadcasting, interrupt-driven execution, etc.

GLONEMO (GLObal NEtwork MOdel) is proposed for obtaining accurate and efficient models of sensor networks, by describing the elements of a sensor network at various levels of abstraction [133]. Several aspects are taken into account, including the hardware that implements a single node, the protocol layers, the application code, energy consumption and the physical environment viewed by sensors. SNs are modeled based on a clean and simple parallel construct at two levels: the physical parallelism between the nodes of the network and the physical or logical parallelism inside a node. The model is based on communicating input/output interpreted automata and is specified as a set of communicating processes written in ReactiveML [104]. GLONEMO has been used to efficiently design the architecture of a global and accurate model of sensor networks. However, the verification of the formal model is unavailable because ReactiveML is not connected to any verification tool.

Therefore, a synchronous language Lustre is introduced to build a global and executable model of a sensor network, including all the elements that influence energy consumption [106]. Lustre is an appropriate language for building such a detailed model, especially when it comes to describing the detailed energy consumption of the hardware and its relationship with time. Further, Lustre allows to build a clean component-based model, provides a modular-abstraction framework, and is connected to various validation tools for formal verification by model checking techniques. Lussensor is a modular model for sensor network systems in Lustre, which serves as a common platform for several abstractions and is essential for connecting to formal validation tools. This model is valid for modeling energy consumption, and contributes to power management of sensor networks and satisfies power constraints.

3.1.2 Calculus

CSN (Calculus for Sensor Networks) [101] is a calculus for sensor networks introduced as a programming model that allows formal verification of the correctness of programs with a global vision of a sensor network application. CSN is devised as a two-layer calculus, offering abstractions for data acquisition, communication, and processing above the link layer of the protocol stack. A sensor network is described by combining a network layer of sensor devices modelled as local objects, and a static type system is introduced to enable safe programming of sensor networks. CSN provides a formal computational programming model for sensor networks. The static type system enables safe programming of sensor networks, allowing for premature detection of certain types of programs that would produce run-time errors. Nevertheless, CSN is still insufficient for analyzing real-world sensor network applications because there are no formal links between the calculus and real-world sensor network programs such as NesC programs.

3.1.3 Timed Formalism

Green et al. demonstrate the formal modeling of a sensor network system for ice formation detection [62]. The evolution of the system is captured by discrete and continuous dynamic models using real-time hybrid automata, i.e. timed automata, implemented using the tool UPPAAL [18]. A sensor network system is modeled as a set of interactive hybrid automata, with the interaction between nodes synchronized on events. The specification for correctness is formulated based on the critical synchronization required among the sensor nodes. Temporal logics are then used for defining properties to verify the system. This approach can be beneficial for building safe systems with high correctness, in that it allows to model and check the system design's correctness prior to implementation through system verification methods.

P. C. Ölveczky et al. present their work to formally model, simulate and model check advanced sensor network algorithms with the rewriting-logic-based Real-Time Maude language and tool [122, 121]. Some general techniques are proposed for modeling and analyzing sensor network algorithms, which are then applied to the modeling, performance estimation, and model checking of the Optimal Geographical Density Control (OGDC) algorithm. Real-Time Maude language extends the object-oriented specification language Maude to support the formal specification and analysis of real-time systems. It is used to model typical sensor network features such as locations, broadcast communication, sensor nodes, time behaviors, etc [121]. Formal model checking in Real-Time Maude allows to reason about both correctness and best-case/worst-case performance. However, the Real-Time Maude model checking analysis of OGDC is incomplete, because it does not analyze all possible behaviors from a given initial sensor network state.

3.1.4 Probabilistic Modeling

A modeling approach tailored to the automatic verification of communication protocols for sensor networks is presented in [13], where a medium access control protocol (S-MAC) for sensor networks is formally modeled and verified by the probabilistic model checker Prism [83]. The main mission of the S-MAC protocol is to reduce energy consumption. A probabilistic model (i.e. Markovian model) of S-MAC is built, including the schedule cordination/maintenance behavior. Relevant probabilistic properties of the protocol are verified, such as probabilistic reachability, time-bounded probabilistic reachability, and expected reachability. Probabilistic properties are expressed in PCTL [70], a temporal logic with probabilistic extensions to CTL [31].

This work contributes to studying the capability of probabilistic model checking techniques for analyzing and verifying important protocols of sensor networks. However, more general techniques are still in need for applying such techniques for more protocols.

3.2 Verification Techniques for TinyOS-based SNs

Since we are interested in formally modeling and verifying TinyOS applications implemented in NesC for sensor network systems, a summary of related work on applying formal methods to TinyOS applications or NesC implementations is presented in this section.

Coleri et al. describe system verification of TinyOS using HyTech, a tool used to check linear hybrid automata [38]. In order to account for the clock cycles required for synchronization events to occur within the operating system that happen instantaneously in the hybrid automata, they implement a dummy state to model the duration of the clock cycles that an event takes. The overhead, therefore, is satisfied within their model by providing additional states. Event and command completion are modeled as additional events because they are across component function calls that return a value of success or failure. In order to simulate packet generation to demonstrate the function of the sensor node interacting on a network, another automaton is added to represent packet generation either from neighboring sensor nodes, or the sensing component of the application. This study achieves verification of the operating system and analyzes power dissipation of a tree-based multi-hop sensor network and demonstrates essential qualities of a sensor network to be considered in design. On one hand, base station connection is essential to the life of the network. On the other hand, the density of the nodes of a sensor network must be inversely proportional to the distance from the base station or energy available must be increased in nodes closest to base station to maintain network functionality because the increase in work of the nodes is proportional to their number of children.

P. Völgyesi et al. [158, 159] propose an interface language, Hierarchical Interface Automata (HIA), for the development of sensor network applications which have a special emphasis on interface specification. HIA extends the Interface Automata [84] formalism, by employing hierarchy to cope with scalability and complexity, and introducing non-preemptable states to reflect the single thread of control of TinyOS applications. Therefore, HIA is able to capture the temporal and typed aspects of interfaces, and support the composition of components. Meanwhile, a modeling environment targeting TinyOS as well as component compatibility rules are presented for developing temporal interface models and checking compatibility for TinyOS components.

A metaprogrammable toolkit, Generic Modeling Environment (GME) [84], is used to create the modeling environment Graphical Development Environment for TinyOS (Gratis). TinyOS concepts such as *interface*, *module*, and *configuration* are specified in a UML class diagram-based notation with OCL constraints [161], and are used as input to GME to generate the meta-model for Gratis. This metamodel defines the mapping of TinyOS concepts to GME concepts. The translator of Gratis is capable of generating NesC programs from graphical models and parsing NesC source code and building corresponding models automatically. As shown in Figure 3.1, two translators are developed in Gratis for checking the composability of components based on their interface models and verifying other properties,

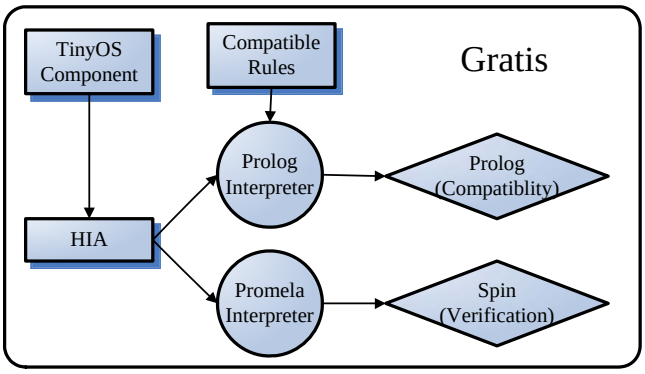


Figure 3.1: The Architecture of Gratis

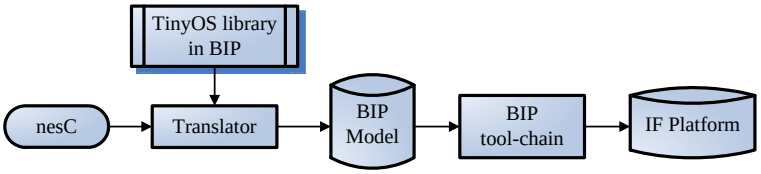


Figure 3.2: The Modeling Flow by BIP

respectively. TinyOS is first modeled as HIAs in Gratis. On one hand, predicates in terms of the logic programming language Prolog are generated by the rules explained in [158] from HIAs, and compatibility between corresponding components can be checked then. On the other hand, each HIA in Gratis is mapped to a Promela proctype/process by transforming automata actions to statements and statement transitions to flow control elements, allowing complex designs and more properties to be verified.

This approach treats TinyOS modules as black box components and is capable of discovering interdependencies of interface primitives and checking compatibility between components. However, it is inadequate to explore the entire state space of the systems. Furthermore, this approach suffers from scalability issues: the composition of n components requires $O(n^2)$ checks among these components.

Basu et al. present their model construction methodology to build heterogeneous real-time systems for TinyOS-based networks using the Behavior-Interaction-Priority (BIP) [14] component framework in [15]. This approach has a self-contained compiler for NesC language and supports both behavioral verification and simulation of sensor network systems. BIP consists of a language for modeling component-based systems and associated execution, simulation and verification tools. As illustrated in Figure 3.2, the BIP models for NesC applications are generated automatically, and are analyzed by powerful state space exploration techniques offered by the IF tool suite [21]. TinyOS is modeled as the composition of two sets of components: one is the set of schedulers for *events* and *tasks*, and the other is the set

of models for hardware components representing *Timer*, *Sensor*, and *Radio*. Two schedulers are used to model the two-level scheduling mechanism of TinyOS, the *Event Scheduler* for the management of events generated by hardware components, and the *Task Scheduler* for executing tasks in *FIFO* order. The global model of the sensor network is composed of BIP components using *connectors*, by specifying interactions between nodes.

This approach models NesC programs as well as the execution platform (i.e. TinyOS) as an abstract machine and provides a global model of the whole sensor network. The framework supports both behavioral verification and simulation of the networked systems. However, the generated model might be incapable of verifying functional behaviors, since computation sequences are ignored to keep model complexity low. This approach is similar to our work, in that it provides automatic generation of formal models from NesC implementations.

Rosa et al. propose an approach for formalizing TinyOS applications with a formal description approach LOTOS [132]. LOTOS is a formal specification language for specifying concurrent and distributed systems, which describes a system through a hierarchy of components or processes (similar to CCS and CSP). The main concepts of NesC, namely *interfaces*, *modules* and *configurations*, are specified as synchronization ports, processes with choices and compositional processes in LOTOS. This formalization of NesC applications emphasizes mostly the interaction between components and allows useful properties of the specification to be checked.

McInnes presents a technique for modeling the concurrency of a TinyOS application using CSP, and using FDR to analyze the resulting process models [107]. This approach contributes to introducing model checking techniques to sensor network systems for enhanced analysis and early error detection and thus increasing reliability and correctness. However, it cannot help application developers avoid the trouble caused by constructing process algebra models entirely manually.

3.3 Domain-specific Tools for SN Verification

In this section, we review existing tools that are dedicated to verifying sensor network systems.

Approaches like SLEDE [69] translate NesC programs into formal description techniques (FDT) like *Promela* and use existing model checkers to conduct verification tasks. SLEDE was proposed for automatic verification of sensor network security protocols implemented in NesC [68, 69, 66, 67]. This approach includes extracting Promela models from protocols implemented in NesC, generating intrusion models and verifying security properties of the models via the state-of-art model checker SPIN. This framework has contributed to decreasing the cost of verification and improving the quality of sensor network security protocol implementations. Our work differs from this approach in several ways. First, we implement

a self-contained parser for the NesC language while SLEDE builds its model extractor on top of NesC compiler [66]. We believe that a self-contained parser would be feasible for optimizing verification tasks. Second, we establish the state space of SNs directly from the source code and thus the state space is more faithful to the source code. Third, our approach targets checking NesC applications, not only security protocol implementations. It is noticed that concurrent processing, failure tolerance, effective communication protocols and other features are critical for sensor network systems. Being aware of the lack of a uniform formal verification framework, we were trying to develop a systematic framework for automatically model checking SNs against properties related to such features.

W. Archer et al. present their work on interface contracts for TinyOS components in [8], which exposes bugs and hidden assumptions caused by improper interface usages, and adds safety conditions to TinyOS applications. Nguyet and Soffa propose to explore the internal structure of SN applications using control flow graphs, but without any error detection [116]. V. Menrad et al. propose to use Statecharts to achieve readable yet more precise formulations of interface contracts [108]. These approaches contribute to the correctness of usages of interfaces, but are incapable of verifying any specific property like safety or liveness.

The tool FSMGen [80] presented by N. Kothari et al. infers compact, user-readable Finite State Machines from TinyOS applications and uses symbolic execution and predicate abstraction for static analysis. This tool captures highly abstract behaviors of NesC programs and has revealed some errors. However, low-level interrupt driven code is not applicable since the tool is based on a coarse approximation of the TinyOS execution model. Some essential features like loops are not supported and the tool provides no support for analyzing the concurrent behaviors of a network (rather than a single sensor).

Anquiro [111] is built based on the Bogor model checking framework [130, 131], for model checking SN software written in the C language for Contiki OS [45]. Source code is 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 a 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 are more likely to cause inaccurate results because it is non-trivial to enforce and prove the soundness of the translation between NesC and FDTs. Hence, approaches for direct verifying NesC programs have been developed.

Werner et al. verify the *ESAWN* protocol by producing abstract behavior models from TinyOS applications, and uses the C model checker CBMC to verify the models [162]. 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 the *ESAWN* protocol and it abstracts away all platform-related behaviors.

Bucur and Kwiatkowska propose Tos2CProver for debugging and verifying TinyOS applications at compile-time, checking memory-related errors and other low-level errors upon registers and peripherals [25, 26, 27]. Tos2CProver translates embedded C code (which is generated by the NesC compiler from NesC source code) to standard C, which is then verified by CBMC. This approach is incapable of checking temporal safety properties like a buffer should be released infinitely often. Moreover, it only checks errors for single-node programs and lacks the ability to find network-level errors. 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 [89] is built upon the simulator TOSSIM [86] for TinyOS applications and uses explicit model checking techniques to verify safety. T-Check checks the execution of SNs by depth-first search (DFS) or random walk to find a violation of user-specified properties, and it adopts stateless and bounded model checking to find bugs. It was used to verify several TinyOS applications and revealed several bugs of components/applications in the TinyOS distribution. One limitation of T-Check is that it does not support the verification of temporal properties. T-Check has implemented a static partial order reduction algorithm, but it requires knowledge of which state transitions are potentially dependent. Furthermore, it only identifies independency at network level, and thus all transitions of the same node are considered as dependent. 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.

3.4 Partial Order Reduction

Partial order reduction (POR) has been proposed for diminishing the state space explosion problem in the procedure of model checking concurrent systems [58, 126, 127, 33]. The key of POR lies in the observation that execution sequences of a concurrent system can be grouped together into equivalent subsets that are indistinguishable by the property being checked. In other words, POR exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders. Applying POR constructs a reduced state space in which each equivalent subset has at least one representative. Initially, POR approaches for checking safety properties including deadlocks were developed [58]. Peled's work in [125] combined POR with on-the-fly model checking while preserving temporal properties. POR has been applied in the verification of industrial concurrent programs, as reported in [60, 112].

Further, POR has been introduced to the verification of systems of different domains, such as timed systems, probabilistic systems, etc. Bengtsson et al. presented a partial-order reduction method for timed systems based on a local-time semantics for networks of timed automata [17]. By letting local clocks in each process advance independently of clocks in other processes and by requiring that two processes re-synchronize their local time scales whenever they communicate, implicit clock synchronization between processes is removed. Further, Minea proposed another POR method that reduces both the number of explored control states and the number of generated time zones [110]. As for probabilistic systems, Argenio and Niebert discussed partial order reduction for probabilistic programs represented as Markov decision processes, preserving probabilistic quantification of reachability properties and maximum and minimum probabilities of LTL-X properties [41]. Moreover, Baier et al. proposed a POR method for probabilistic branching time [10] and Giro et al. revised the POR for LTL properties applied to probabilistic model checking [56].

Farzan and Meseguer proposed a method to apply POR without changing the underlying model checking algorithms, by rewriting semantics [48]. In this way, POR could be achieved for any language with relatively little effort. Experiments with JVM and a Promela-like language have indicated that significant state space reductions and time speedups could be obtained by this approach.

Kurshan et al. presented a POR method that allows to generate a smaller number of choices from each state at compile time [82]. Such POR methods are referred to as static POR, and it has been shown that static POR can be directly implemented with existing verification tools without modifying the verification engine, and may be applied with other techniques like symbolic model checking and localization reduction to obtain a combined reduction.

Brim et al. combined POR with distributed memory state exploration which extends the computational power for LTL model checking [22]. Flanagan and Godefroid initially explored an arbitrary interleaving of the various concurrent processes/threads and dynamically tracked interactions between them to identify backtracking points where alternative paths in the state space need to be explored [50]. This approach is called dynamic POR (DPOR) and is developed for stateless model checking techniques. The authors also showed that DPOR significantly reduces the search space of multi-threaded programs even when traditional POR techniques are helpless. To solve the inefficiency of stateless runtime model checking, Yang et al. proposed a stateful dynamic partial order reduction algorithm (SDPOR) to avoid a potential unsoundness when DPOR is naively applied in the context of stateful model checking [164]. This approach combines light-weight state recording with DPOR, and strikes a good balance between state recording overheads and the elimination of redundant searches. Their experiments confirm the effectiveness of the approach on several multi-threaded benchmarks in C. Yang et al. proposed a practical algorithm for distributed dynamic POR (DDPOR) to help distributed Inspect, the distributed version of the model

checker Inspect, gain nearly linear speedup on realistic examples [163]. This work speeds up stateless model checking using computer clusters and get benefits from DPOR [165], achieving linear speedup on multi-threaded programs.

Gueta et al. proposed a cartesian semantics that reduces the amount of nondeterminism in concurrent programs by delaying unnecessary context switches [65]. A novel dynamic POR algorithm, i.e., cartesian partial order reduction, was then developed using this semantics and preliminary experimental results showed a significant potential saving in the number of explored states and transitions.

There is also work in combining POR with other reduction techniques such as symmetry reduction [47]. Traditionally, POR was applied with enumerative depth-first search for the analysis of asynchronous network protocols, and symbolic model checking was mainly applied to the analysis of synchronous hardware designs. Alur et al. combined both techniques and developed a method for using POR techniques in symbolic BDD-based invariant checking [7]. Kahlon et al. proposed Monotonic Partial Order Reduction (MPOR) which effectively combines dynamic POR with symbolic state space exploration for model checking concurrent software [77]. This approach adopts quasi-monotonic sequences of thread-ids, which can be used both for explicit or symbolic model checking. By restricting the interleavings explored to the set of quasi-monotonic sequences, MPOR achieves automatic pruning of redundant interleavings in a SAT/SMT solver based search for symbolic model checking, with the guarantee of both soundness and optimality.

In terms of program verification, POR has been widely used [16]. Naumovich et al. proposed a POR method for Ada programs based on FLAVERS [115]. FLAVERS is a general data flow based verification approach and has been applied to Ada [46], Java [114], C++, etc. The famous model checker for Java programs Java PathFinder also adopts POR to improve the performance of verification [157]. Message Passing Interface (MPI) is the dominant model used in high performance computing (HPC) nowadays. Given the distributed memory nature of MPI programs, POR has considerable potential for alleviating state space explosion, and Palmer et al. discussed a dynamic POR method for MPI-based parallel programs [123]. The formal semantics for a non-trivial subset of MPI was defined, and then independence theorems are proved in order to obtain a semantically clear and general partial-order reduction approach for MPI. The approach describes the exact dependencies between MPI non-blocking send operations and their tests for completion for the first time. Kundu et al. proposed the tool Satya which combines static and dynamic POR techniques with SystemC semantics to achieve scalable testing of SystemC TLM designs [81].

Among the approaches for SN verification discussed in Section 3.3, only two of them are using POR to prune the search space. The first is T-Check [89], which adopts stateless model checking with a static POR method. The POR implemented here only checks the independence among sensors, which means that intra-sensor concurrency is not reduced. The other is Tos2CProver [27], which adopts the POR approach proposed in [34] to reduce

unnecessary interleaving among interrupt handlers. Since Tos2CProver only handles individual sensor programs, concurrency between sensors is beyond its scope. Both approaches are limited to checking safety properties only.

3.5 Summary

The features of NesC and TinyOS presented in Chapter 2 have shown that it is challenging to analyze NesC programs (or TinyOS applications) for sensor networks. As shown in this chapter, a number of approaches have been proposed for modeling, analyzing and even verifying sensor network systems. These approaches have helped to improve the quality of sensor network systems and contributed to the research community in various ways. Nevertheless, one major limitation among them that they only formalize NesC/TinyOS concepts rather than semantics. This means that they are limited to high-level analysis and are incapable of directly analyzing programs.

One aim of this thesis is to develop a methodology for establishing verification techniques directly upon NesC programs. In Chapter 4, we discuss our early-stage work on a translation-based approach on model checking sensor networks. Then in Chapter 5, we present the formalization of NesC programs, the TinyOS execution model and sensor network composition, which facilitates the *direct* analysis and verification of SNs.

Chapter 4

Translation-based Verification of SNs

In this chapter, we propose to automatically generate STCSP (Stateful Timed CSP) models [148, 143] from sensor network systems implemented in NesC. Section 4.1 relates the TinyOS execution model to STCSP semantics; Section 4.2 explains how STCSP models are constructed from NesC programs; Section 4.3 evaluates the performance of NesC2STCSP with several examples; and Section 4.4 briefly summarizes this chapter. In this chapter, we use the TinyOS application presented in Example 6 as a running example throughout the discussion of the translation approach.

Example 6 (The Blink Application)**.** *Here we present a simple NesC program Blink. The source code of Blink is shown in Figure 4.1, which contains two NesC files, BlinkAppC.nc (Figure 4.1(a)) and BlinkC.nc (Figure 4.1(b)). In this application, an alarm is set at every second to toggle the red LED (i.e., led0) of the sensor.*

4.1 STCSP Semantics for TinyOS

The formal definition of STCSP models is presented in [143], where an STCSP model is defined as a 3-tuple $\mathcal{R} = (Var, init, P)$. *Var* is a set of global variables, *init* is the initial valuation of the variables and P is the running STCSP process. A brief introduction of the syntax and the informal semantics of STCSP processes can be found in Appendix C. In this chapter, we use the following definitions to present the STCSP semantics of the TinyOS execution model.

- $Ch(c, n)$ defines a channel named c with a buffer of size n and a channel with a buffer of zero size requires synchronous input and output.
- $c!v$ is a channel output operation which sends the data v to the channel c.
- $c?x$ is a channel input operation which reads from the channel c and caches the data in the local variable x, while $c?v$ is a conditional channel input that only reads the channel when the data to be read match v.

```
1  #include <Timer.h>
2  configuration BlinkAppC
3  {
4  }
5  implementation
6  {
7    components MainC, BlinkC;
8    components LedsC;
9    components new TimerMilliC();
10   BlinkC -> MainC.Boot;
11   BlinkC.Timer0 -> TimerMilliC;
12   BlinkC.Leds -> LedsC;
13 }
```

(a) Program BlinkAppC.nc

```
1  module BlinkC ()
2  {
3    uses interface Timer<TMilli> as Timer0;
4    uses interface Leds;
5    uses interface Boot;
6  }implementation{
7    event void Boot.booted(){
8      call Timer0.startPeriodic(1000);
9    }
10   event void Timer0.fired(){
11     call Leds.led0Toggle();
12   }
13 }
```

(b) Program BlinkC.nc

Figure 4.1: Source code of the Blink application

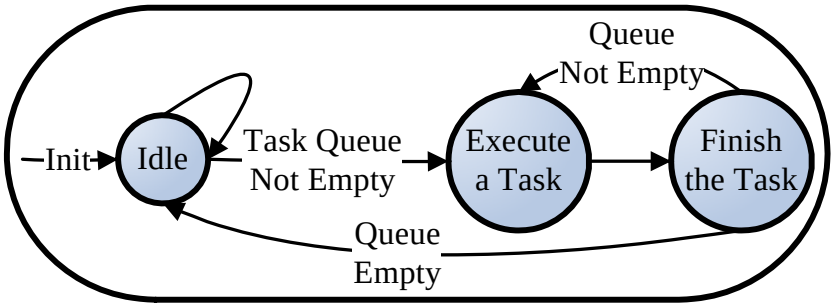


Figure 4.2: Task scheduler model

- $Const(x, v)$ defines a constant variable x with the value v;

The execution model of TinyOS is based on split-phase operations, run-to-completion *tasks* and *interrupt handlers*. First, we model the task scheduler that schedules the execution of tasks in a certain order, and in our work, we assume it to be the FIFO order. Then the task scheduler can be represented as the state machine shown in Figure 4.2, based on which, we then formalize the task scheduler as Definition 1.

Definition 1 (Task Scheduler)**.** *The task scheduler sdl of TinyOS is modeled as an STCSP process* $P_{sdl} = if(Q_t \neq \varnothing)\{Q_t?tsk \rightarrow sdl!tsk \rightarrow sdl?EOT \rightarrow P_{sdl}\}else\{P_{sdl}\}$*, where:*

- $Q_t = Ch(Q_t, n_t)$ *is a queue for deferred tasks and* $n_t > 0$*;*
- $sdl = Ch(sdl, 0)$ *is a synchronous channel for notifying the task tsk to execute;*
- EOT *is a constant variable with the unique value* 0 ($Const(EOT, 0)$) *denoting the end of a task.*

The process P_{sdl} is blocked until the task queue Q_t is not empty, and a task *tsk* is fetched from the queue and the corresponding task is informed to run via the channel *sdl*.

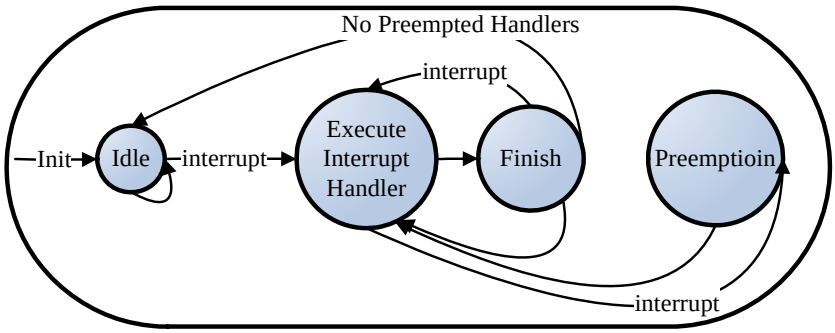


Figure 4.3: Interrupt manager model

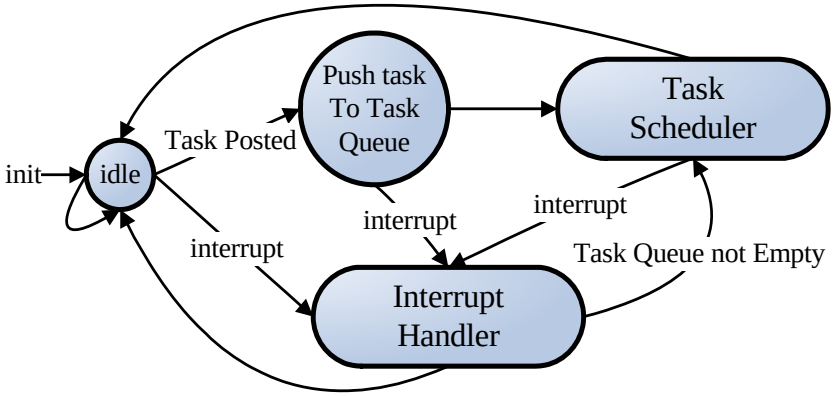


Figure 4.4: The Execution Model of TinyOS

The process that models a task *tsk* is blocked until the scheduler process P_{sdl} sends the data *tsk* to the channel *sdl*, which will be shown in next Section. After the execution of the task, the task process writes the channel *sdl* with EOT to inform P_{sdl} of the task *tsk*'s completion, and P_{sdl} will start to execute the next task in Q_t.

In contrast to tasks, an interrupt can preempt tasks or other interrupts. The interrupt manager is introduced to manage the executions and interactions between interrupt handlers. In the interrupt manager, a new interrupt preempts the execution of the current one. The state machine of the interrupt manager is shown in Figure 4.3. Since interrupts can preempt the execution of tasks, the global state machine of the execution model of TinyOS should capture such features, as presented in Figure 4.4.

Since there is no appropriate interrupt semantics in STCSP, an alternative is to model interrupt handlers and the task scheduler to concurrently execute. In the following, we present the STCSP model of an interrupt handler.

Definition 2 (Interrupt Handler)**.** *The interrupt handler ih of an interrupt itr is modeled as an STCSP process* $P_{ih} = ifb(x_{itr} == ON)\{P'_{ih} \rightarrow Q_t!tsk_{ih} \rightarrow \{x_{itr} = OFF\} \rightarrow P_{ih}\}$*, where:*

- *ifb is an STCSP construct to define a conditional process which is blocked until the condition is satisfied;*
- x_{itr} *is a status variable indicating whether the interrupt itr is activated;*
- P'_{ih} *denotes the processing of the interrupt handler;*
- *and* $Q_t!tsk_{ih}$ *enqueues a task to signal the corresponding event of the interrupt.*

Example 7 (One-shot Timing Interrupts)**.** *The one-shot timer implemented by TimerMilliC is modeled as the following process:*
$P_{Timer\_Os} = ifb(TimerMilliC\_Timer\_os == ON)\{Wait[TimerMilliC\_period];$
$Qt!tsk_{TimerMilliC\_fired} \rightarrow \{TimerMilliC\_Timer\_os = OFF\} \rightarrow P_{Timer\_Os}\};$
where the variable TimerMilliC_Timer_os is the status denoting whether the one-shot timer is active.

Example 8 (Periodic Timing Interrupts)**.** *In the modeling of periodic timer, the status of the interrupt is never disabled in the interrupt process but it can be disable by calling a stop command. The following is the process of periodic timing interrupts:*
$P_{Timer\_Prd} = ifb(TimerMilliC\_Timer\_prd == ON)\{Wait[TimerMilliC\_period];$
$Qt!tsk_{TimerMilliC\_fired} \rightarrow P_{Timer\_Prd}\}.$

The following presents the formal definition of the TinyOS execution model.

Definition 3 (Execution Model)**.** *The TinyOS execution model is a 6-tuple* $(Var, Q_t, \mathcal{S}, \mathcal{A}, s_0, \delta)$*, consisting of:*

- Var*: a set of variables declared in the NesC program, the status variables of interrupts and all channels except the task queue* Q_t*;*
- Q_t*: a queue for scheduling tasks;*
- $\mathcal{S}$*: a set of reachable states;*
- $\mathcal{A}$*: a set of possible actions;*
- s_0*: an initial state; and*
- $\delta : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$*: a transition function.*

4.2 Generation of STCSP Model

TinyOS provides a component library which encapsulates operating system environment and hardware functionalities as predefined interfaces and components. And this is required when running NesC applications. Therefore, a set of STCSP models representing the TinyOS

NesC Elements	STCSP Constructs
Split-phase operations	Processes communicating via synchronous channels.
Component	Global variables, processes modeling behaviors of commands, events, and tasks.
System	The interleaving composition of all processes.

Table 4.1: The Mapping Algorithm

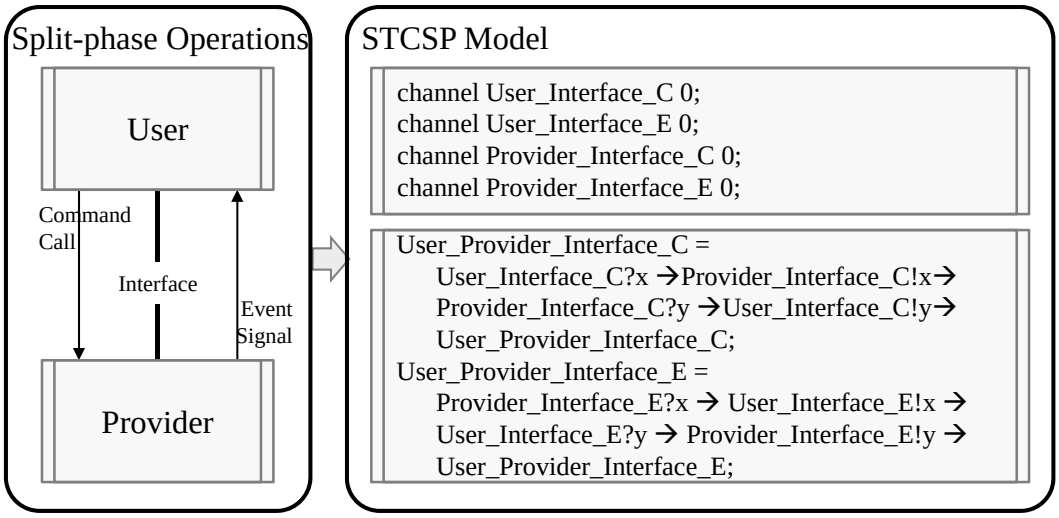


Figure 4.5: Modeling Split-phase Operations

component library is statically provided (such as Timer, Sensor, LED, etc.). These static models are used when applying the mapping algorithm for the generation of STCSP models.

As shown in Table 4.1, the mapping algorithm for extracting STCSP models from NesC code consists of three steps: modeling split-phase operations, modeling components and modeling the application as a system-level process. Each step is explained in detail as follows.

Step 1: model split-phase operations

As shown in Figure 4.5, components interact with each other via interfaces. A wiring in NesC "wires" a component to another by an interface and the two components are required to use and provide the interface respectively. Wiring is modeled by defining two processes transferring command calls and event signals between the user component and the provider component.

Definition 4 (Wiring Expression)**.** *A wiring expression user.ui → prov.pi is modeled as an STCSP process* $P_w = (P_{user\_ui\_prov\_pi_c} \;|||\; P_{user\_ui\_prov\_pi_e})$*, where* $P_{user\_ui\_prov\_pi_c} =$

$(user\_ui_c?cmd.args.Req \rightarrow prov\_pi_c!cmd.args.Req \rightarrow prov\_pi_c?cmd.Rep \rightarrow user\_ui_c!cmd.Rep \rightarrow P_{user\_ui\_prov\_pi_c})$, *and* $P_{user\_ui\_prov\_pi_e} = (prov\_pi_e?evt.args.Req \rightarrow user\_ui_e!evt.args.Req \rightarrow user\_ui_e?evt.Rep \rightarrow prov\_pi_e!evt.Rep \rightarrow P_{user\_ui\_prov\_pi_e})$, *where:*

- *Req is a constant denoting the request of calling a command or signaling an event;*
- *Rep is a constant denoting the completion of the requested command or event;*
- $user\_ui_c$ *is a synchronous channel for the user component to send a request when it calls a command of the interface ui;*
- $prov\_pi_c$ *is a synchronous channel to which the provider component listens for the invocation of a command of the interface pi from the user;*
- $prov\_pi_e$ *is a synchronous channel for the provider component to send a request when it signals an event of the interface pi; and*
- $user\_ui_e$ *is a synchronous channel to which the user component listens for the invocation of an event of the interface ui from the provider.*

Example 9 (Modeling a wiring expression)**.** *The wiring expression* $BlinkC.Timer0 \rightarrow TimerMilliC$ *in Figure 4.1(a) is translated into the following process:*

$P_{BlinkC\_Timer0\_TimerMilliC} = P_C \;|||\; P_E$;

$P_C = BlinkC\_Timer0\_C?x0.x1.Req \rightarrow TimerMilliC\_Timer\_C!x0.x1.Req \rightarrow$
$TimerMilliC\_Timer\_C?x0.Rep \rightarrow BlinkC\_Timer0\_C!x0.Rep \rightarrow P_C$;

$P_E = TimerMilliC\_Timer\_E?x0.Req \rightarrow BlinkC\_Timer0\_E!x0.Req \rightarrow$
$BlinkC\_Timer0\_E?y0.Rep \rightarrow TimerMilliC\_Timer\_E!y0.Rep \rightarrow P_E$.

The intuition in modeling a wiring expression is to direct the command calling from a user to the corresponding provider that implements the command and vice versa for event signalings.

Step 2: model a component

A component is separated into three parts: component variables, commands, events and tasks. Each function, which is a task, command or event, is specified as a process, and a component is the interleaving composition of all the processes representing its implemented functions, as shown in Figure 4.6. The statements in a function are modeled as STCSP events or processes, as shown in Table 4.2. To emphasize, a task is scheduled in a task scheduler and does not execute immediately once it is posted. The queue Q_t is defined for scheduling task, and the channel *sdl* is used for communication between tasks and the task scheduler, as shown in Definition 1.

Example 10 (Modeling a command call)**.** *The statement call* $Timer0.startPeriodic(1000)$ *in Figure 4.1(b) is modeled as the following STCSP construct:*

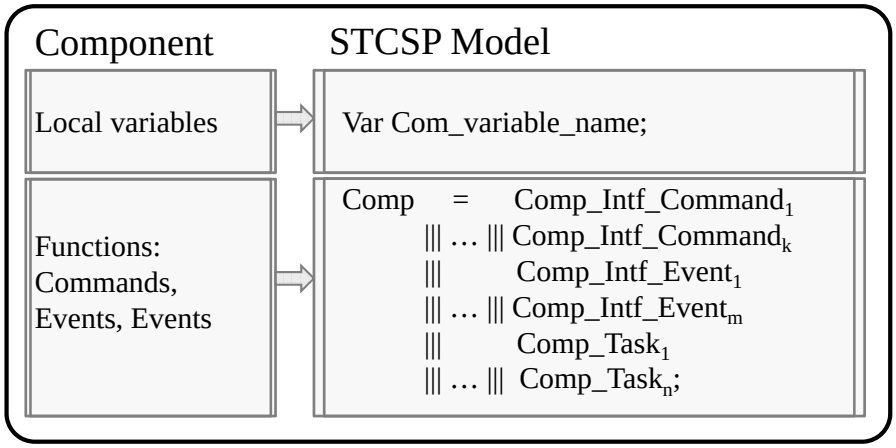


Figure 4.6: Modeling a component

$BlinkC\_Timer0\_C!stPrd.1000.Req \rightarrow BlinkC\_Timer0\_C?stPrd.Rep$,
where stPrd is the constant defined to identify the startPeriodic command for the Timer interface, 1000 is the parameter of the command calling, Req denotes the request of the command and Rep denotes the completion of the command.

Definition 5 (Command Implementation)**.** *A command comp.i.cmd of the interface i implemented by the component comp is modeled as an STCSP process* $P_{comp\_cmd\_i} = comp\_i_c?cmd.args.Req \rightarrow Cmd \rightarrow comp\_i_c!cmd.Rep \rightarrow P_{comp\_cmd\_i}$, *where Cmd is an STCSP process constructed by translating the statements of the command body.*

Example 11 (Modeling a command)**.** *The command Timer.startPeriodic(dt) of the component TimerMilliC of Example 6 is modeled as the following STCSP process:*
$P_{TimerMilliC\_Timer\_stPrd} = TimerMilliC\_Timer\_C?stPrd.dt.Req \rightarrow$
$\{ TimerMilliC\_period = dt; \} \rightarrow \{ TimerMilliC\_Timer\_prd = ON; \}$
$\rightarrow TimerMilliC\_Timer\_C!stPrd.Rep \rightarrow P_{TimerMilliC\_Timer\_stPrd}$, *where stPrd is the identifier of the command startPeriodic, TimerMilliC_period is a component variable of the component TimerMilliC and TimerMilliC_Timer_prd is the status variable of the periodic timer associated to the component TimmerMilliC.*

Definition 6 (Event Implementation)**.** *An event comp.i.evt of the interface i implemented by the component comp is modeled as an STCSP process* $P_{comp\_i\_evt} = comp\_i_e?\ evt.args.Req \rightarrow Evt \rightarrow comp\_i_e!evt.Rep \rightarrow P_{comp\_i\_evt}$, *where Evt is an STCSP process constructed by translating the statements of the event implementation.*

Example 12 (Modeling an event)**.** *The Timer0.fired event in Figure 4.1(b) can be modelled as the following STCSP process:*
$P_{BlinkC\_Timer0\_fired} = BlinkC\_Timer0\_E?fr.Req \rightarrow BlinkC\_Leds\_C!t0.Req \rightarrow$

NesC statements	STCSP processes
user.ui → prov.pi	$(user\_ui_c?cmd.args.Req \rightarrow prov\_pi_c!cmd.args.Req \rightarrow prov\_pi_c?cmd.Rep \rightarrow user\_ui_c!cmd.Rep) \mid\mid\mid (prov\_pi_e?evt.args.Req \rightarrow user\_ui_e!evt.args.Req \rightarrow user\_ui_e?evt.Rep \rightarrow prov\_pi_e!evt.Rep)$
(user) call i.cmd()	$user\_i_c!cmd.args.Req \rightarrow user\_i_c?cmd.Rep$
comp.i.cmd{CMD}	$comp\_i_c?cmd.args.Req \rightarrow CMD \rightarrow comp\_i_c!cmd.Rep$
(prov) signal i.evt()	$user\_i_e!evt.args.Req \rightarrow user\_i_e?evt.Rep$
comp.i.evt{EVT}	$comp\_i_e?evt.args.Req \rightarrow EVT \rightarrow comp\_i_e!evt.Rep$
task scheduler	$P_{sdl} = if(Q_t \neq \varnothing)\{Q_t?tsk \rightarrow sdl!tsk \rightarrow sdl?EOT \rightarrow P_{sdl}\}else\{P_{sdl}\}$
post tsk()	$if(!Q_t.Has(tsk))\{Q_t!tsk \rightarrow Skip\}else\{Skip\};$
tsk{TSK}	$sdl?tsk \rightarrow TSK \rightarrow sdl!EOT$
interrupt handler ih	$P_{ih} = ifb(x_{itr} == ON)\{P'_{ih} \rightarrow Q_t!tsk_{ih} \rightarrow \{x_{itr} = OFF\} \rightarrow P_{ih}\}$
if(B){A}else{C}	$if\ (B)\{A\}else\{C\};$
while(B){A}	$if(B)\{A;\ WHILE\}else\{Skip\};$
do{A}while(B)	$A;\ \ if(B)\{WHILE\}else\{Skip\};$
for(A;B;C){D}	$A;\ \ ReFor;(ReFor\ =\ if(B)\{D;\ C;\ ReFor\ \}else\{Skip\};)$

Table 4.2: STCSP processes for NesC constructs

$BlinkC\_Leds\_C?t0.Rep \rightarrow BlinkC\_Timer0\_E!fr.Rep \rightarrow P_{BlinkC\_Timer0\_fired}$, *where fr is the identifier of the command fired and* $t0$ *is the identifier of the event* $led0Toggle$.

Definition 7 (Task)**.** *A task tsk is modeled as an STCSP process* $P_{tsk} = sdl?tsk \rightarrow Tsk;\ sdl!EOT$, *where Tsk is an STCSP process constructed by translating the statements of the task.*

Example 13 (Modeling Timer Fired Task)**.** *The fired task implemented by TimerMilliC is task void TimerFired(){signal Timer.fired(); }. By Definition 7, it can be modelled as the following STCSP process:*

$P_{TimerMilliC\_TimerFired} = sdl?tsk_{TimerMilliC\_TimerFired} \rightarrow TimerMilliC\_Timer\_E!fr.Req \rightarrow TimerMilliC\_Timer\_E?fr.Rep \rightarrow sdl!EOT.$

Based on the semantics of STCSP models, we can construct the execution trace of the event *Boot.booted* as STCSP events as shown in Example 14.

Example 14 (Executing Trace)**.** *The execution trace of the event Boot.booted of Example 6 is:* $BlinkC\_Timer0\_C.stPrd.1000.Req \rightarrow TimerMilliC\_Timer\_C.stPrd.1000.Req \rightarrow (startPrd's\ body) \rightarrow TimerMilliC\_Timer\_C.stPrd.Rep \rightarrow BlinkC\_Timer0\_C.stPrd.Rep.$

Step 3: build the system-level process

After specifying all components and their wirings, a top-level process is defined as the interleaving of all existing processes in runtime to represent the whole system, including the process modeling the task scheduler.

$$System \mathrel{\widehat{=}} P_{sdl} \;|||\; P_{wire_1} \;|||\; \ldots \;|||\; P_{wire_w} \;|||\; P_{ih_1} \;|||\; \ldots \;|||\; P_{ih_i} \;|||\; P_{comp_1} \;|||\; \ldots \;|||\; P_{comp_k};$$

In the above definition of the STCSP process *System*, $P_{wire_j}(1 < j < w)$ is the process modeling the wire expression $wire_j$ and w is the number of wiring expression defined in the application. Similarly, $P_{ih_j}(0 < j < i)$ and $P_{comp_j}(0 < j < k)$ model the interrupt handler ih_j and the component $comp_j$, respectively.

Modeling timed constraints TinyOS maintains middlewares such as Timer and Alarm to support the real-time features for developing NesC applications. The timing middlewares of TinyOS are modeled as timed processes with timed operators including *Wait*, *Deadline*, *Waituntil*, *Interrupt* and so on, as illustrated in [148]. These models are implemented statically as part of the runtime environmental library when automatically constructing the STCSP models in our framework.

4.3 Experiments and Evaluation

The generated STCSP models are then verified by the model checker PAT [144], which supports the verification of STCSP models [148, 143] against properties such as non-termination (represented as deadlock freeness of STCSP models), state reachability, temporal properties. This approach is named NesC2STCSP and has been integrated in our framework NesC@PAT. In this section, we illustrate the execution of the NesC2STCSP framework with the Blink application presented in Example 6, in which an LED is turned on and off periodically. We also run the application BlinkC' for comparison, in which two timers are used to toggle two LEDs periodically. Moreover, we run the Sensing application, which periodically samples data and toggles a certain LED based on the value of the data sensed. Our testbed is a PC with the Intel Core i7-2640M CPU at 2.80GHz and 7.88GB RAM.

During model extraction, timed operator *Wait* is used when modeling the command *startPeriodic* of the interface *Timer*. The generated STCSP model consists of variables, channels and processes. Three properties are verified against for both the Blink and the Blink' applications. The first is the goal of non-termination; the second is the goal that the timer(s) is(are) fired infinitely often; and the third is the goal that the LED should eventually be toggled whenever its corresponding timer is fired. As for the Sensing application, three properties are defined as well. The first is non-termination, the second is that the timer can be fired infinitely often, and the third is that whenever the timer is fired some data can be eventually sampled.

The results of the verification of the three applications are shown in Table 4.3. These

System(LoC)	Assertion	Result	# States	Time (s)
Blink (18)	Non-termination	✓	41	0.01
	□◇ TimerFired	✓	86	0.01
	□ TimerFired → ◇ led0Toggled	✓	44	0.01
Blink' (24)	Non-termination	✓	243	0.02
	□◇TimerFired	✓	706	0.08
	□ TimerFired → ◇ led0Toggled	✓	357	0.05
Sensing (29)	Non-termination	✓	1048K	60
	□◇TimerFired	✓	4313K	369
	□ TimerFired → ◇ led0Toggled	✓	2598	239

Table 4.3: Verification Results of NesC2STCSP

Category	Features		Sup
Inherited from C	Pointer		×
	Struct		×
	Data Types	32-bit integer	✓
		Boolean	✓
		Float	×
		Double	×
		8/16/64-bit (un)signed integer	×
	Algebraic operations	$+, -, \times, /, \%$	✓
	Logical operations	$\&\&, \|\|, \neg, =, \neq, <, \leq, >, \geq$	✓
	Bitwise operations	$\&, \|, \sim, \wedge, >>, <<$	×
	Label statement		×
NesC specific	Generic components and interfaces		×
	Fan-in(out) wirings		×
	Atomic statement		×

Table 4.4: Summary of syntax supported by NesC2STCSP

examples show that our approach is useful and effective for verifying *TinyOS* applications. However, to verify the Sensing application takes much more time than to verify either the Blink application or the Blink' application. This is because that the Sensing application has more sources of concurrency than the other two applications, including the concurrency between the timer and the sensing device, and the concurrency caused by the non-determinism of the value of data being sensed.

4.4 Summary

This chapter presents the NesC2STCSP approach on the automatic generation of STCSP models from sensor network programs in NesC. As shown in Table 4.4, where **Sup** denotes

whether the corresponding language feature is supported or not, only a subset of NesC syntax is supported by NesC2STCSP. Most syntax features are not supported due to the limitation of the modeling language STCSP.

- Since STCSP only supports integer and boolean data types, C features such as pointers, structs, data types except integer or boolean are not supported. Furthermore, generic components or interfaces are not supported because they use data types as parameters in their definitions of components or interfaces.
- Bitwise operations are not supported because they are not allowed in STCSP.
- Label statements are not supported because there is no similar semantic operator in STCSP and thus it is complicated to model label statements in STCSP. Similarly, atomic statements that disable interrupts, and fan-in or fan-out features of multiple-to-one wiring expressions are not supported.

We believe that this approach contributes to the robustness of sensor network systems because it allows NesC programmers to model check their code without being troubled by manually constructing formal models. However, due to the semantic differences between NesC and STCSP, the STCSP models obtained from NesC programs are inevitably large, complex and redundant.

Intuitively, direct verification is straightforward and more efficient as specialized optimization techniques are possible. However, direct verification is infeasible without a complete operational semantics of the NesC language. The idea is to define the operational semantics of NesC, and explore the system state space based on them to for verification purposes. The challenge is that to define the operational semantics is not an easy task, since currently there exists no formal description of TinyOS or NesC. Since the execution model of TinyOS application is well-defined, we can develop a formal description of this execution model and then operational semantics can be defined further. This formalization approach will be discussed in the next chapter.

Chapter 5

Direct Verification of SNs

This chapter presents our work on directly modeling and verifying SNs and is organized as follows. Section 5.1 presents the formal semantics of the NesC language; Section 5.2 formalizes the TinyOS execution model; Sections 5.3 and 5.4 discuss the network composition semantics and the modeling of the environment of SNs, respectively; Section 5.5 discusses the model checking algorithms adopted in our work; Section 5.6 evaluates our approach with experiments; and Section 5.7 summarizes the chapter.

5.1 Formal Semantics of NesC

In this section, we formalize the semantics of the NesC language. In particular, the formal definitions of sensors in the LTS semantics are given in Section 5.1.1, and the operational semantics of NesC language constructs is illustrated in Section 5.1.2.

5.1.1 Definitions

In this thesis, we focus on sensors running TinyOS applications implemented in NesC. The behaviors of a sensor are then determined by the scheduler of tasks and the concurrent execution between tasks and device interrupts.

Definition 8 (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 records posted tasks in a certain order; R is a buffer that keeps incoming messages sent by other sensors; init is the initial valuation of the variable set A; and P is a program, composed by the running NesC program M interrupted by various devices H, i.e., $P = M \triangle H$.*

H models (and often abstracts) the behaviors of hardware devices such as Timer, Receiver and Reader (i.e. the sensing device). Because tasks are deferred computations, when posted they are pushed into the task queue T for scheduling in a certain order. By default, tasks would be scheduled in FIFO order on TinyOS. However, TinyOS allows programmers to

configure the scheduler with a different order. In our work, we assume that tasks are always scheduled in FIFO order. We remark that T and R are empty initially for any sensor model $\mathcal{S}$. The interrupt operator ($\triangle$) is introduced to capture the interrupt-driven feature of the TinyOS execution model, which will be explained later in Section 2.3.

The variables in A are categorized into two groups. One is composed of variables declared in the NesC program, which are further divided into two categories according to their scopes, i.e. component variables and local variables. Component variables are defined in a component's scope, whereas local variables are defined within a function's or a statement block's scope. All component variables are loaded to the variable set A at initialization. As for local variables, they are loaded on demand at run time. To avoid naming conflicts, the name of each variable is first prefixed with the component name. A local variable is renamed with the line number of its declaration position in addition. The other is a set of *device variables* that capture the status of hardware devices. For example, *MessageC.Status* is introduced to model the status of the messaging device. A status variable is added into A after the compilation if the corresponding device is accessed in the program.

Assume that a sensor executes *TrickleAppC* implemented for Example 1. By Definition 8, the corresponding sensor model is $\mathcal{S} = (A, T, R, init, P)$. In *TrickleAppC*, component *TrickleC* is referred as *App*, so *App* is used for renaming its variables. The variable set after renaming is $A = \{$ *MessageC.Status*, *App.summary*, *App.code*, *App*.34.*pkt*, $\cdots\}$, where variables with two-field names (e.g. *App.summary*) are component variables except for *MessageC.Status* (a device status variable), and those with three-field names (e.g. *App*.34.*pkt*) are local variables. *init* is the initial valuation where *MessageC.Status* = *OFF*, *App.summary* = 0, *App.code* = 0, *App*.34.*pkt* = *null*, $\cdots$. In *TrickleAppC*, only the messaging device *MessageC* is accessed. Therefore, initially the program P is event *Boot.booted* (denoted by M_b) interrupted by the messaging device (denoted by *MessageC*), i.e., $P = M_b \triangle$ *MessageC*. Event *Boot.booted* is implemented by *TrickleC* as follows.

```
event void Boot.booted(){
      call AMControl.start(); //Start the messaging device
}
```

Calling *AMControl.start* will execute the corresponding command body in component *ActiveMessageC*. Component *ActiveMessageC* is defined in the TinyOS component library for activating the messaging device. When *AMControl.start* is completed, the event *AMControl.startDone* (implemented by *TrickleC*) will be signaled. If *AMControl.start* returns *SUCCESS*, the function *sendSummary* is called for sending the summary. Otherwise, the command *AMControl.start* is re-called, as shown in Figure 5.1. We use a model *MessageC* to describe the behaviors of the messaging device of a sensor, as will be presented in Section 5.2.1. If *AMControl.start* is performed successfully, the program P of $\mathcal{S}$ will become $M_b' \triangle$ *MessageC*$'$ where M_b' and *MessageC*$'$ are the new program status and the new device status after exe-

```
1 event void AMControl.startDone(error_t rs){
2     if (rs! = SUCCESS){
3         //The previous request fails
4         call AMControl.start();
5     } else {
6         sendSummanry();
7      }
8 }
```

Figure 5.1: Event *AMControl.startDone*

cuting the command respectively, and the value of *MessageC.Status* will be modified. □

Definition 9 (Sensor State)**.** *Let $\mathcal{S} = (A, T, R, init, P)$ be a sensor model. A sensor state C of $\mathcal{S}$ is a tuple (V, Q, B, P) where V is the valuation of variables A; Q is a sequence of tasks, being the content of T; B is a sequence of messages, being the content of R; P is the executing program.*

For a sensor model $\mathcal{S} = (A, T, R, init, P)$, its initial state is $C^{init} = (init, \varnothing, \varnothing, P)$, in respect that initially task queue T and message buffer R are empty. A transition is written as $(V, Q, B, P) \xrightarrow{e} (V', Q', B', P')$ (or $C \xrightarrow{e} C'$ for short), where e is a text label assigned to the transition to denote the behavior of the transition. Next, we define the behavior of a sensor as an LTS.

Definition 10 (Sensor Transition System)**.** *Let $\mathcal{S} = (A, T, R, init, P)$ be a sensor model. The transition system of $\mathcal{S}$ is defined as a tuple $\mathbb{T} = (\mathbb{C}, init, \rightarrow)$, where $\mathbb{C}$ is the set of all reachable sensor states and $\rightarrow$ is the transition relation.*

The transition relation is defined as $\rightarrow: \mathbb{C} \times \sum \times \mathbb{C}$, where $\sum$ is the set of possible labels. And $\rightarrow$ is formally obtained through a set of firing rules associated with each and every NesC programming construct.

5.1.2 Operational Semantics

Let $\mathcal{S}(A, T, R, init, P)$ be a sensor model. The transition system of $\mathcal{S}$ is $\mathbb{T}(\mathbb{C}, init, \rightarrow)$, obtained by the firing rules of P. The transition relation $(\rightarrow)$ is formally defined through a set of firing rules associated with each and every NesC programming constructs, as follows.

- $Function(E)$ returns the first function contained in expression E.
- τ is a label denoting a silent transition.
- ✓ simply denotes the termination of the execution of a statement.
- e is used to denote the label of a transition.

- x is a mathematical expression and a variable is considered as an expression.
- $E[a/x]$ *updates* the expression E by replacing the sub-expression x with a.
- $Prog(L)$ represents the program point that is labelled by L.
- $V(x)$ is a function over expressions and it *evaluates* the value of the expression x in the valuation V.
- $V[v/x]$ *updates* the expression x with the value v evaluated in the valuation V.
- I is a status variable, denoting whether interrupts are allowed at a certain state.
- ac is a status variable, serving as the counter of active *atomic* blocks. This variable is introduced to support nested *atomic* blocks, as shown in the rules for *atomic* statements (rules $atm1$-$atm5$).
- $at\{P\}$ is introduced to encode the list of statements P within an *atomic* block, where interrupts are disabled.
- $^\frown$ is sequence concatenation.
- $FstFnc(L_{arg})$ returns the first argument in L_{arg} which contains function calls.
- ϵ simply means null.
- $Impl(f, L_{arg})$ returns the body of function f with arguments L_{arg}.
- $\{F\}$ denotes the list of statements F within a function's scope.
- $L_{arg}[a'/a]$ replaces argument a with a' in the argument list L_{arg}.
- r is a status variable, denoting the value returned by a certain function.
- $\langle P \llcorner Q \lrcorner \rangle$ is introduced as the intermediate representation of a loop statement, where P is the list of statements in the current iteration and Q is the original loop statement.
- *Stop* denotes the termination of a sensor.

Expression (E), defined by rules $expr1$, $expr2$ and $expr3$.

A purely mathematical expression refers to an expression composed of variables and constant values, without any function calls. For example, $a \times 10 - b$ is a purely mathematical expression. The firing rule for this case is presented in the rule $expr1$. The rule $expr1$ is used in other rules later to check if an expression contains an incomplete function call.

$$\frac{Function(E) = \epsilon}{(V,\ Q,\ B,\ E) \xrightarrow{\tau} (V,\ Q,\ B,\ \checkmark)} \quad [\ expr1\ ]$$

If an expression contains some function call, it will firstly rewrite the original expression with the target program of the function call (rule *expr*2). We remark that the function body is executed step by step. For example, expression $a \times sum(b, 20)$ will execute the function call $sum(b, 20)$ at first.

$$\frac{P = Function(E) \neq \epsilon,\ (V,\ Q,\ B,\ P) \xrightarrow{e} (V',\ Q',\ B',\ P')}{(V,\ Q,\ B,\ E) \xrightarrow{e} (V',\ Q',\ B',\ E[P'/P])} \quad [\ expr2\ ]$$

If the function call completes its execution, it will update the original expression with the returned value $V'(r)$, as shown in the rule *expr*3. For example, when $sum(b, 20)$ finishes its execution, the original expression $a \times sum(b, 20)$ will be updated with its return value v_r, becoming $a \times v_r$. The rule *expr*3 restricts that the execution of the *return* statement and the updating of the expression E should be considered as one single atomic event. Therefore, exactly one status variable r is sufficient to correctly capture the semantics of function calls even if there are concurrent functions, nested function calls, or multiple function calls. This will be elaborated later in the rules for function calls.

$$\frac{P = Function(E) \neq \epsilon,\ (V,\ Q,\ B,\ P) \xrightarrow{e} (V',\ Q',\ B',\ \checkmark)}{(V,\ Q,\ B,\ E) \xrightarrow{e} (V',\ Q',\ B',\ E[V'(r)/P])} \quad [\ expr3\ ]$$

Example 15 (LTS of an expression)**.** *Given an expression involving a method call as shown in Figure 5.2, the method call will be executed first and the corresponding LTS is presented in Figure 5.3. At state C_2, the expression will become $(i + 5)$ where 5 is the value returned by the method call.*

```
1 i + min(10, 5);
2
3 int min(int a, int b){
4     if(a < b)
5         return a;
6     else return b;
7 }
```

Figure 5.2: An expression

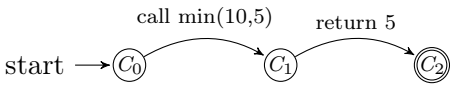


Figure 5.3: LTS of the expression

Assignment $(x := E)$, defined by rules *as*1 and *as*2.

If the expression is purely mathematical, denoted by $(V,\ Q,\ B,\ E) \xrightarrow{\tau} (V,\ Q,\ B,\ \checkmark)$, the variable x will be updated immediately with the value of E (i.e., $V(E)$), as shown in the rule *as*1. The label e of an assignment transition is constructed in the form of $s.assign.x.v$, where s is the identifier of the sensor that runs the current assignment, *assign* is a constant element, x is the variable being updated and v is the new value of x after the assignment. For example, when executing $count := 1$ for sensor s_1, the label of the corresponding transition

is $e = s_1.assign.count.1$.

$$\frac{(V,\ Q,\ B,\ E) \xrightarrow{\tau} (V,\ Q,\ B,\ \checkmark),\ v = V(E),\ e = s.assign.x.v}{(V,\ Q,\ B,\ x := E) \xrightarrow{e} (V[v/x],\ Q,\ B,\ \checkmark)} \quad [\ as1\ ]$$

Rule $as2$ presents the case when the expression E contains a function call. The expression E is executed first by applying the rule $expr2$. Upon the return statement of the function call, the rule $expr3$ is applied, which will replace the function call with the return value and then the rule $as1$ could be applied in the next step to update the variable x. For example, $count := count + getCount()$ will first execute the function call $getCount()$.

$$\frac{(V,\ Q,\ B,\ E) \xrightarrow{e} (V',\ Q',\ B', E'),\ E' \neq \checkmark}{(V,\ Q,\ B,\ x := E) \xrightarrow{e} (V',\ Q',\ B',\ x := E')} \quad [\ as2\ ]$$

Example 16 (LTS of an assignment)**.** *Given an assignment as shown in Figure 5.4, suppose that initially $v = 0$ and $k = 3$ and let (v, k) denote each state. The corresponding LTS is presented in Figure 5.5.*

```
1 v = k + min(10, 5);
2
3 int min(int a, int b){
4     if(a < b)
5         return a;
6     else return b;
7 }
```

Figure 5.4: Assignment

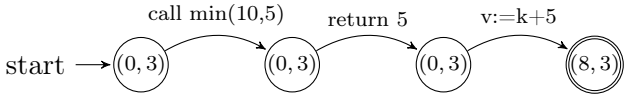


Figure 5.5: LTS of the assignment

Post $(post\ tsk())$, defined by rules $post1$ and $post2$.

TinyOS does not allow duplicated tasks in the task queue, and thus to execute a *post* statement requires checking if the task is already in the task queue. If there are no duplicated tasks, then the task will be enqueued immediately (rule $post1$); otherwise the task is dropped (rule $post2$).

$$\frac{e = s.post.t,\ t \notin Q,\ Q' = Q \frown \langle t \rangle}{(V,\ Q,\ B,\ post\ tsk()) \xrightarrow{e} (V,\ Q',\ B,\ \checkmark)} \quad [\ post1\ ]$$

$$\frac{e = s.post.t.fail,\ t \in Q}{(V,\ Q,\ B,\ post\ tsk()) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark)} \quad [\ post2\ ]$$

Command call $(call\ intf.cmd(L_{arg}))$, defined by rules $call1$ and $call2$.

In the statement *call intf.cmd*(L_{arg}), L_{arg} is the list of arguments. As shown by rule *call*1, if the arguments contain no function calls, the execution of a command call will transit directly to the execution of the corresponding command body. The label of the transition in the rule *call*1 is constructed as *call.intf.cmd* where *call* is a keyword denoting a command call, *intf* is the name of the interface of the command and *cmd* is the name of the command.

$$\frac{e = call.intf.cmd, FstFnc(L_{arg}) = \epsilon,\ F = Impl(intf.cmd, L_{arg})}{(V,\ Q,\ B,\ call\ intf.cmd(L_{arg})) \xrightarrow{e} (V,\ Q,\ B,\ \{F\})} \quad [\ call1\ ]$$

If the arguments involve function calls, the function calls will be executed first (rule *call*2). In the rule *call*2, the actual execution of the command call (i.e., the rule *call*1) is delayed by some function invoked in the arguments. And thus the label of the transition is identical to the one obtained by executing the functions, i.e., e from $(V,\ Q,\ B,\ a) \xrightarrow{e} (V',\ Q',\ B',\ a')$.

$$\frac{FstFnc(L_{arg}) = a,\ a \neq \epsilon,\ (V,\ Q,\ B,\ a) \xrightarrow{e} (V',\ Q',\ B',\ a')}{(V,\ Q,\ B,\ call\ intf.cmd(L_{arg})) \xrightarrow{e} (V',\ Q',\ B',\ call\ intf.cmd(L_{arg}[a'/a]))} \quad [\ call2\ ]$$

Event signal (*signal intf.evt*(L_{arg})), defined by rules *sig*1 and *sig*2.

In the statement *signal intf.evt*(L_{arg}) , L_{arg} is the list of arguments. As shown by rule *sig*1, if the arguments contain no function calls, the execution of an event signal will transit directly to the execution of the corresponding event body. Otherwise, the function calls in the arguments will be executed first (rule *sig*2). The labels of both rules are constructed in the way similar to the rules *call*1 and *call*2.

$$\frac{e = s.sig.intf.evt, FstFnc(L_{arg}) = \epsilon,\ F = Impl(intf.evt, L_{arg})}{(V,\ Q,\ B,\ signal\ intf.evt(L_{arg})) \xrightarrow{e} (V,\ Q,\ B,\ \{F\})} \quad [\ sig1\ ]$$

$$\frac{FstFnc(L_{arg}) = a,\ a \neq \epsilon,\ (V,\ Q,\ B,\ a) \xrightarrow{e} (V',\ Q',\ B',\ a')}{(V,\ Q,\ B,\ signal\ intf.evt(L_{arg})) \xrightarrow{e} (V',\ Q',\ B',\ signal\ intf.evt(L_{arg}[a'/a]))} \quad [\ sig2\ ]$$

Function invocation (*func*(L_{arg})), defined by rules *inv*1 and *inv*2.

In the statement *func*(L_{arg}) , L_{arg} is the list of arguments. As shown by rule *inv*1, if the arguments contain no function calls, the execution of a function invocation will transit directly to the execution of the corresponding function body. Otherwise, the function calls in the arguments will be executed first (rule *inv*2).

$$\frac{e = s.invoke.func, FstFnc(L_{arg}) = \epsilon,\ F = Impl(func, L_{arg})}{(V,\ Q,\ B,\ func(L_{arg})) \xrightarrow{e} (V,\ Q,\ B,\ \{F\})} \quad [\ inv1\ ]$$

$$\frac{FstFnc(L_{arg}) = a,\ a \neq \epsilon,\ (V,\ Q,\ B,\ a) \xrightarrow{e} (V',\ Q',\ B',\ a')}{(V,\ Q,\ B,\ func(L_{arg})) \xrightarrow{e} (V',\ Q',\ B',\ func(L_{arg}[a'/a]))}\ [\ inv2\ ]$$

Return (*return* or *return* E), defined by rules *return*1, *return*2 and *return*3.

A return statement is only legal within a function's scope, and it will terminate the execution of the function body ($\{return;\ F\}$) it belongs to, no matter if there is any subsequent statement. If a return statement has no returned value (i.e. *return*), the function body is simply terminated (rule *return*1). Otherwise, either the rule *return*2 or the rule *return*3 will be applied, depending on whether the returned expression (E) contains any function call.

$$\frac{}{(V,\ Q,\ B,\ \{return;\ F\}) \xrightarrow{s.return} (V[\epsilon/r],\ Q,\ B,\ \checkmark)}\ [\ return1\ ]$$

$$\frac{(V,\ Q,\ B,\ E) \xrightarrow{e} (V',\ Q',\ B',\ E'),\ E' \neq \checkmark}{(V,\ Q,\ B,\ \{return\ E;\ F\}) \xrightarrow{e} (V',\ Q',\ B',\ \{return\ E';\ F\})}\ [\ return2\ ]$$

$$\frac{(V,\ Q,\ B,\ E) \xrightarrow{\tau} (V,\ Q,\ B,\ \checkmark),\ v_r = V(E)}{(V,\ Q,\ B,\ \{return\ E;\ F\}) \xrightarrow{s.return.v_r} (V[v_r/r],\ Q,\ B,\ \checkmark)}\ [\ return3\ ]$$

Example 17 (LTS of *return* statement)**.** *Given a return statement as shown in Figure 5.6, suppose that initially $v_r = 0$ and let (v_r) denotes each state. The corresponding LTS is presented in Figure 5.7.*

```
1 ...
2 return 100+sum(5, 10);
3 ...
4
5 int sum(int x, int y){
6     return x + y;
7 }
```

Figure 5.6: *return* statement

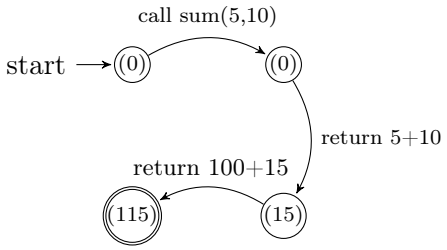


Figure 5.7: LTS of the *return* statement

Go-to Statement (*gt* L), defined by the rule *gt*.

During the parsing stage, our tool generates a table of labels and their corresponding program point. In the execution of a go-to statement, $Prog(L)$ returns the program point of the label L by looking at this table. Moreover, in C, a go-to statement provides an unconditional jump to a labeled statement in the same function. The NesC language follows this convention for go-to statements. Therefore, a go-to statement will only affects the

program point within the function it belongs to. We use {}, the boundary of functions, to capture this feature.

$$\frac{P_L = Prog(L)}{(V,\ Q,\ B,\ \{gt\ L;\ P\}) \overset{s.gt.L}{\rightarrow} (V,\ Q,\ B,\ \{P_L\})} \quad [\ gt\ ]$$

Sequential composition $(P;\ Q)$, defined by rules $seq1$ and $seq2$.

The operator ; describes the sequential composition of statements, and is defined as left-associate.

$$\frac{(V,\ Q,\ B,\ P) \overset{e}{\rightarrow} (V',\ Q',\ B',\ P'),\ P' \neq \checkmark}{(V,\ Q,\ B,\ P;\ S) \overset{e}{\rightarrow} (V',\ Q',\ B',\ P';\ S)} \quad [\ seq1\ ]$$

$$\frac{(V,\ Q,\ B,\ P) \overset{e}{\rightarrow} (V',\ Q',\ B',\ \checkmark)}{(V,\ Q,\ B,\ P;\ S) \overset{e}{\rightarrow} (V',\ Q',\ B',\ S)} \quad [\ seq2\ ]$$

Atomic $(atomic\{P\})$, defined by rules $atm1$, $atm2$, $atm3$, $atm4$ and $atm5$.

In an atomic block, interrupts are disabled. Thus, the status variable I will be set as *off* whenever the system enters a new *atomic* block (rule $atm1$). Meanwhile, the atomic counter ac is increased by 1. The construct $at\{P'\}$ is used to denote the statements to be executed while I remains *off*.

$$\frac{(V,\ Q,\ B,\ P) \overset{e}{\rightarrow} (V',\ Q',\ B',\ P'),\ P' \neq \checkmark,\ v_{ac} = V(ac) + 1}{(V,\ Q,\ B,\ atomic\{P\}) \overset{e}{\rightarrow} (V'[off/I, v_{ac}/ac],\ Q',\ B',\ at\{P'\})} \quad [\ atm1\ ]$$

If the *atomic* block terminates in one step, I or ac will not be changed (rule $atm2$), because after that step the *atomic* block terminates immediately. This rule is designed to avoid redundant states caused by an unnecessary $at\{\}$ block.

$$\frac{(V,\ Q,\ B,\ P) \overset{e}{\rightarrow} (V',\ Q',\ B',\ \checkmark)}{(V,\ Q,\ B,\ atomic\{P\}) \overset{e}{\rightarrow} (V',\ Q',\ B',\ \checkmark)} \quad [\ atm2\ ]$$

The rule $atm3$ describes the execution of the remaining statements of an *atomic* block after it disables interrupts.

$$\frac{(V,\ Q,\ B,\ P) \overset{e}{\rightarrow} (V',\ Q',\ B',\ P'),\ P' \neq \checkmark}{(V,\ Q,\ B,\ at\{P\}) \overset{e}{\rightarrow} (V',\ Q',\ B',\ at\{P'\})} \quad [\ atm3\ ]$$

The rules $atm4$ and $atm5$ handle the last statement of an *atomic* block. On one hand, the atomic counter ac is decreased by 1. On the other hand, interrupts are enabled only

when there are no active *atomic* blocks (rule $atm5$).

$$\frac{(V,\ Q,\ B,\ P) \xrightarrow{e} (V',\ Q',\ B',\ \checkmark),\ v_{ac} = V(ac) - 1 > 0}{(V,\ Q,\ B,\ at\{P\}) \xrightarrow{e} (V'[v_{ac}/ac],\ Q',\ B',\ \checkmark)} \quad [\ atm4\ ]$$

$$\frac{(V,\ Q,\ B,\ P) \xrightarrow{e} (V',\ Q',\ B',\ \checkmark),\ v_{ac} = V(ac) - 1 = 0}{(V,\ Q,\ B,\ at\{P\}) \xrightarrow{e} (V'[on/I, 0/ac],\ Q',\ B',\ \checkmark)} \quad [\ atm5\ ]$$

Function $(\{F\})$, defined by rules $func1$ and $func2$.

$\{F\}$ is used to denote the execution of a function, where $\{\}$ denotes the boundary of the function, and F denotes the statements of the function to be executed. If the function body finishes its execution, the whole function terminates (rule $func2$). Note that the function boundary ($\{\}$) disappears when rule $func2$ is applied.

$$\frac{(V,\ Q,\ B,\ F) \xrightarrow{e} (V',\ Q',\ B',\ F'),\ F' \neq \checkmark}{(V,\ Q,\ B,\ \{F\}) \xrightarrow{e} (V',\ Q',\ B',\ \{F'\})} \quad [\ func1\ ]$$

$$\frac{(V,\ Q,\ B,\ F) \xrightarrow{e} (V',\ Q',\ B',\ \checkmark)}{(V,\ Q,\ B,\ \{F\}) \xrightarrow{e} (V',\ Q',\ B',\ \checkmark)} \quad [\ func2\ ]$$

If-Else $(if(c)\ P\ else\ S)$, defined by rules $if1$, $if2$ and $if3$.

A statement without the *else* clause like $if(c)P$ is considered as an if-else statement with S being $\checkmark$, i.e., $if(c)\ P\ else\ \checkmark$. If the condition has some function call, the system will execute the function call before evaluating the condition (rule $if1$).

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{e} (V',\ Q',\ B',\ c'),\ c' \neq \checkmark}{(V,\ Q,\ B,\ if(c)\ P\ else\ S) \xrightarrow{e} (V',\ Q',\ B',\ if(c')\ P\ else\ S)} \quad [\ if1\ ]$$

The statement $if(c)\ P\ else\ S$ will behave as P if the condition c is evaluated as *True* (rule $if2$); otherwise, it behaves as S (rule $if3$).

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{\tau} (V,\ Q,\ B,\ \checkmark),\ V(c) = True}{(V,\ Q,\ B,\ if(c)\ P\ else\ S) \xrightarrow{e} (V,\ Q,\ B,\ P)} \quad [\ if2\ ]$$

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{\tau} (V,\ Q,\ B,\ \checkmark),\ V(c) = False}{(V,\ Q,\ B,\ if(c)\ P\ else\ S) \xrightarrow{e} (V,\ Q,\ B,\ S)} \quad [\ if3\ ]$$

While $(while(c)\ P)$, defined by rules $wh1$, $wh2$, $wh3$, $wh4$, $wh5$ and $wh6$.

The statement *do* P *while*(c) is considered as the sequential composition of P and a while statement, i.e., P; *while*(c) P. Similarly to if-else statements, if the condition has some function call, the system will execute the function call before evaluating the condition (rules $wh1$ and $wh2$).

As shown in the sequential composition rule $seq2$, a statement is dismissed after it completes its execution. This prevents a loop statement like *while* from repeating the execution of the loop because both the loop condition and the loop body are missing after the first iteration of the loop. Therefore, the construct $\llcorner W_0 \lrcorner$ is introduced to cache the original *while* statement including the original loop condition and loop body, in order to allow repeated execution of the loop.

$$\frac{W_0 = \mathit{while}(c)\ P,\ (V,\ Q,\ B,\ c) \xrightarrow{e} (V',\ Q',\ B',\ c'),\ c' \neq \checkmark}{(V,\ Q,\ B,\ W_0) \xrightarrow{e} (V',\ Q',\ B',\ \mathit{while}(c')\ P \llcorner W_0 \lrcorner)} \quad [\ wh1\ ]$$

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{e} (V',\ Q',\ B',\ c'),\ c' \neq \checkmark}{(V,\ Q,\ B,\ \mathit{while}(c)\ P \llcorner W_0 \lrcorner) \xrightarrow{e} (V',\ Q',\ B',\ \mathit{while}(c')\ P \llcorner W_0 \lrcorner)} \quad [\ wh2\ ]$$

In the rules $wh3$ and $wh4$, $\langle\rangle$ is used to represent the iterative execution of a loop statement, and $\llcorner W_0 \lrcorner$ is included to cache the original while loop statement. The firing rules of $\langle P \llcorner S \lrcorner \rangle$ will be defined later by rules $loop1$ and $loop2$.

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{\tau} (V,\ Q,\ B,\ \checkmark),\ V(c) = \mathit{True}}{(V,\ Q,\ B,\ \mathit{while}(c)\ P) \xrightarrow{e} (V,\ Q,\ B,\ \langle P \llcorner \mathit{while}(c)\ P \lrcorner \rangle)} \quad [\ wh3\ ]$$

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{\tau} (V,\ Q,\ B,\ \checkmark),\ V(c) = \mathit{True}}{(V,\ Q,\ B,\ \mathit{while}(c)\ P \llcorner W_0 \lrcorner) \xrightarrow{e} (V,\ Q,\ B,\ \langle P \llcorner W_0 \lrcorner \rangle)} \quad [\ wh4\ ]$$

If the condition c is evaluated to be *False*, the *while* statement will terminate immediately, as shown in the rules $wh5$ and $wh6$.

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark),\ V(c) = \mathit{False}}{(V,\ Q,\ B,\ \mathit{while}(c)\ P) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark)} \quad [\ wh5\ ]$$

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark),\ V(c) = \mathit{False}}{(V,\ Q,\ B,\ \mathit{while}(c)\ P \llcorner W_0 \lrcorner) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark)} \quad [\ wh6\ ]$$

For $(\mathit{for}(R;\ \ c;\ \ S)\ P)$, defined by rules $for1$, $for2$, $for3$, $for4$, $for5$, $for6$, $for7$ and $for8$.

In a *for* statement, the statements before evaluating the condition will be executed

exactly once, as shown in the following rules *for*1 and *for*2.

$$\frac{(V,\ Q,\ B,\ R) \xrightarrow{e} (V',\ Q',\ B',\ R'),\ R' \neq \checkmark}{(V,\ Q,\ B,\ for(R;\ \ c;\ \ S)\ P) \xrightarrow{e} (V',\ Q',\ B',\ for(R';\ \ c;\ \ S)\ P)} \quad [\ for1\ ]$$

$$\frac{(V,\ Q,\ B,\ R) \xrightarrow{e} (V',\ Q',\ B',\ \checkmark)}{(V,\ Q,\ B,\ for(R;\ \ c;\ \ S)\ P) \xrightarrow{e} (V',\ Q',\ B',\ for(\epsilon;\ \ c;\ \ S)\ P)} \quad [\ for2\ ]$$

When there are no statements to be executed before evaluating the condition, the condition c will be executed if it contains any function call, as in the rules *for*3 and *for*4.

$$\frac{F_0 = for(\epsilon;\ \ c;\ \ S)\ P,\ (V,\ Q,\ B,\ c) \xrightarrow{e} (V',\ Q',\ B',\ c'),\ c' \neq \checkmark}{(V,\ Q,\ B,\ F_0) \xrightarrow{e} (V',\ Q',\ B',\ for(\epsilon;\ \ c';\ \ S)\ P\ \llcorner F_0 \lrcorner)} \quad [\ for3\ ]$$

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{e} (V',\ Q',\ B',\ c'),\ c' \neq \checkmark}{(V,\ Q,\ B,\ for(\epsilon;\ \ c;\ \ S)\ P\ \llcorner F_0 \lrcorner) \xrightarrow{e} (V',\ Q',\ B',\ for(\epsilon;\ \ c';\ \ S)\ P\ \llcorner F_0 \lrcorner)} \quad [\ for4\ ]$$

Example 18 (LTS of *for* statement)**.** *Given a for statement as shown in Figure 5.8, suppose that initially* $sum = 0$ *and let* (i, sum) *denotes each state. The corresponding LTS is presented in Figure 5.9.*

```
1 for(i = 0; i < 4; i++){
2       sum = sum+i;
3 }
```

Figure 5.8: *for* statement

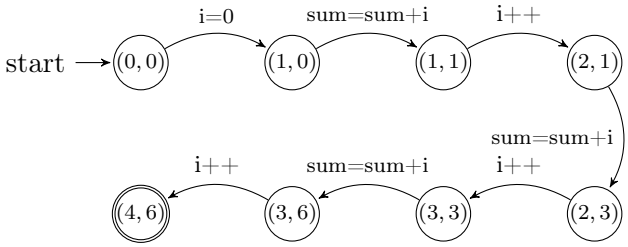


Figure 5.9: LTS of the *for* statement

When the condition is evaluated as *True*, the *for* statement transits to a loop $\langle P;\ \llcorner for(\epsilon;\ \ c;\ \ S)\ P \lrcorner \rangle$ (rules *for*5 and *for*6); otherwise, the *for* statement terminates immediately as shown in the rules *for*7 and *for*8.

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark),\ V(c) = True}{(V,\ Q,\ B,\ for(\epsilon;\ \ c;\ \ S)\ P) \xrightarrow{e} (V,\ Q,\ B,\ \langle P;\ S\ \llcorner for(\epsilon;\ \ c;\ \ S)\ P \lrcorner \rangle)} \quad [\ for5\ ]$$

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark),\ V(c) = True}{(V,\ Q,\ B,\ for(\epsilon;\ \ c;\ \ S)\ P\ \llcorner F_0 \lrcorner) \xrightarrow{e} (V,\ Q,\ B,\ \langle P;\ S\ \llcorner F_0 \lrcorner\rangle)} \quad [\ for6\ ]$$

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark),\ V(c) = False}{(V,\ Q,\ B,\ for(\epsilon;\ \ c;\ \ S)\ P) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark)} \quad [\ for7\ ]$$

$$\frac{(V,\ Q,\ B,\ c) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark),\ V(c) = False}{(V,\ Q,\ B,\ for(\epsilon;\ \ c;\ \ S)\ P\ \llcorner F_0 \lrcorner) \xrightarrow{e} (V,\ Q,\ B,\ \checkmark)} \quad [\ for8\ ]$$

Loop $(\langle P\ \llcorner S \lrcorner\rangle)$, defined by rules *loop*1, *loop*2 and *loop*3.

A loop $\langle P\ \llcorner S \lrcorner\rangle$ describes the iterative execution of *while* (*do-while*) statements and *for* statements, where P is the current iteration and S caches the original loop statement including the original looping condition. The statement P is executed step by step (rule *loop*1).

$$\frac{(V,\ Q,\ B,\ P;\ P_1) \xrightarrow{e} (V',\ Q',\ B',\ P'),\ P' \neq \checkmark,\ P \neq gt\ L}{(V,\ Q,\ B,\ \langle P;\ P_1\ \llcorner S \lrcorner\rangle) \xrightarrow{e} (V',\ Q',\ B',\ \langle P'\ \llcorner S \lrcorner\rangle)} \quad [\ loop1\ ]$$

The rule *loop*2 is introduced to support labelled break and continue statements. There is no explicit function boundary {} in this rule, which means that the loop and the label is within the scope of the same function. This rule implies that a go-to statement in a function invoked by a loop statement will not affect the loop itself. For example, given an intermediate program $\langle\{gt\ L;\ P_0\}P\ \llcorner S \lrcorner\rangle$, after executing $gt\ L$ the program will become $\langle\{P_L\}P\ \llcorner S \lrcorner\rangle$.

$$\frac{P_L = Prog(L)}{(V,\ Q,\ B,\ \langle gt\ L;\ P\ \llcorner S \lrcorner\rangle) \xrightarrow{s.gt.L} (V,\ Q,\ B,\ P_L)} \quad [\ loop2\ ]$$

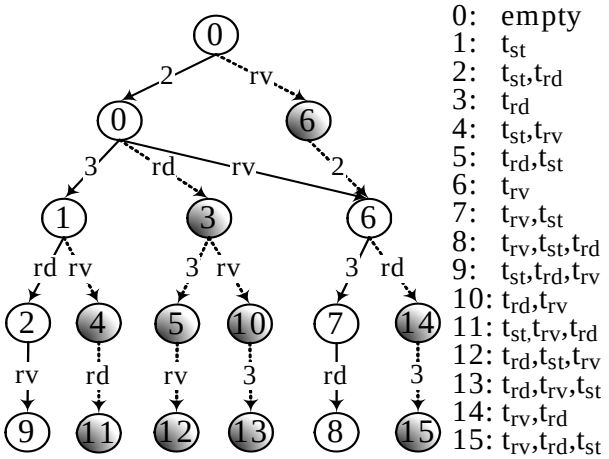
When P terminates, the loop body of the original statement (S) will be executed (rule *loop*3).

$$\frac{(V,\ Q,\ B,\ P) \xrightarrow{e} (V',\ Q',\ B',\ \checkmark)}{(V,\ Q,\ B,\ \langle P\ \llcorner S \lrcorner\rangle) \xrightarrow{e} (V',\ Q',\ B',\ S)} \quad [\ loop3\ ]$$

Although the semantics of a loop seems the same as that of the sequential composition of statements, this structure is necessary in order to define the semantics of *continue* and *break* statements in loops.

```
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) Application Code



(b) State Space of Boot.booted

Figure 5.10: Example Code

Continue (*continue*), defined by the rule *cont*.

A *continue* statement is only legal in a loop, i.e., $\langle P \llcorner S \lrcorner \rangle$. When a *continue* statement is encountered, the current iteration will be skipped immediately and a new iteration will be started (rule *cont*).

$$\frac{}{(V,\ Q,\ B,\ \langle \mathit{continue};\ P \llcorner S \lrcorner \rangle) \overset{s.\mathit{continue}}{\rightarrow} (V,\ Q,\ B,\ S)} \quad [\ \mathit{cont}\ ]$$

Break (*break*), defined by the rule *brk*.

Similarly, a *break* statement is only legal in a loop, i.e., $\langle P \llcorner S \lrcorner \rangle$. If a *break* statement is encountered in a loop, the loop terminates immediately (rule *brk*).

$$\frac{}{(V,\ Q,\ B,\ \langle \mathit{break};\ P \llcorner S \lrcorner \rangle) \overset{s.\mathit{break}}{\rightarrow} (V,\ Q,\ B,\ \checkmark)} \quad [\ \mathit{brk}\ ]$$

Example 19 (State Space of Functions)**.** *In the code shown in Figure 5.10(a), 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 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, the state graph of event Boot.booted is shown in Figure 5.10(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 Figure 5.10(b),*

Operation	Component	Remark
Activation	ActiveMessageC	Activate the messaging device by command $SplitControl.start()$.
Message Transmission	AMSenderC	Send a message by the command $AMSend.send(dst, msg, length)$.
Message Reception	AMReceiverC	Signal its *receive* event if the destination address of the incoming message is the local ID **or** the broadcast ID.
	AMSnooperC	Signal its *receive* event if the destination address of the incoming message is **neither** the local ID **nor** the broadcast ID.
	AMSnoopingReceiverC	Always signal its *receive* event regardless of the destination address of the incoming message.

Table 5.1: Components of the Messaging Device

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 6).

5.2 Formalization of TinyOS Execution Model

In this section, we tackle the formalization of features and components related to the TinyOS execution model, including the formalization of interrupts, the formal models of hardware devices, and the interrupt operator which models the current execution among interrupts and tasks.

5.2.1 Hardware Model Collection

The models of the hardware devices are developed systematically. According to the TinyOS component library, we develop a component model library for common-used components like *AMSenderC*, *AMReceiverC*, *TimerC*, etc. Currently, our approach models the messaging device, timing device and sensing device as follows.

Messaging device. In the TinyOS component library, components *ActiveMessageC*, *AMSenderC*, *AMReceiverC*, *AMSnooperC* and *AMSnoopingReceiverC* are designed for different operations on the messaging device, such as device activation, message transmission and message reception. The descriptions of the primary components for the messaging device are presented in Table 5.1. In the following we present the semantics of the operations or interrupt handlers of the messaging device.

```
1 task void t_sd(){
2     //update device status
3     signal AMSend.sendDone();
4 }
5 task void t_rcv(){
6     //update device status
7     signal Receive.receive();
8 }
```

Figure 5.11: Completion Tasks

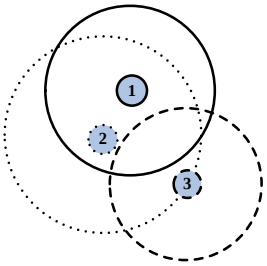


Figure 5.12: An SN Example

- Message transmission. The semantic structure *Send* is defined to model the behaviors of sending a message. *Send* is defined as $(s, dst, amid, msg)$, where s is the identifier of the sensor which sends a message, dst is the destination, $amid$ is the AM type of the channel and msg is the message itself. The task t_{sd} is the *sendDone* task of the sender component $SenderC_{amid}$ related to the channel $amid$ and here we use $SendDone(s, amid)$ to return the *sendDone* task of the sensor s corresponding to the channel $amid$. The task t_{sd} is abstracted by the way explained in Section 2.3, as shown in Figure 5.11. The message sent msg is stored in a buffer of the component $SenderC_{amid}$, which is later used as a parameter in the event signalling of *sendDone* event in the task t_{sd}.

$$\frac{t_{sd} = SendDone(s, amid),\ t_{sd} \notin Q,\ Q' = Q \frown \langle t_{sd} \rangle}{(V,\ Q,\ B,\ Send(s, dst, amid, msg)) \overset{s.send.dst.amid.msg}{\rightarrow} (V,\ Q',\ B,\ \checkmark)} \ [\ send\ ]$$

- Message reception. The semantic structure *Rcv* is defined to model the behaviors of receiving a message. When an incoming message arrives, a task (t_{rcv}) will be enqueued at the task queue, as shown in the rule *rcv*. The function $AMID(msg)$ returns the channel $amid$ which msg is sent through. The task t_{rcv} is the *receive* task of the receiver component $ReceiverC_{amid}$ related to the channel $amid$ and here we use $Receive(s, amid)$ to return the *receive* task of the sensor s when receiving a message from the channel $amid$. The task t_{rcv} is abstracted by the way explained in Section 2.3, as shown in Figure 5.11. The received message msg is stored in a buffer of the component $ReceiverC_{amid}$, which is later used as a parameter in when signalling the *receive* event in the task t_{rcv}.

$$\frac{B = \langle msg \rangle^{\frown} B',\ amid = AMID(msg),\ t_{rcv} = Receive(s, amid),\ t_{rcv} \notin Q,\ Q' = Q \frown \langle t_{rcv} \rangle}{(V,\ Q,\ B,\ Rcv) \overset{s.rcv.amid.msg}{\rightarrow} (V,\ Q',\ B',\ Rcv)} \ [\ rcv\ ]$$

Timing device. The *Timer* and *Alarm* interfaces of TinyOS declare commands for the timing device. And the most important and common commands are command *startOneShot* and *startPeriodic*. The command *startOneShot* sets a single-shot timer to some time units

in the future, while *startPeriodic* sets a periodic timer to repeat at every interval. The rule *frd* is defined for the firing of timers.

$$\frac{t_{frd} \notin Q,\ Q' = Q \frown \langle t_{frd} \rangle}{(V,\ Q,\ B,\ Fired()) \overset{s.fired}{\rightarrow} (V,\ Q',\ B,\ \checkmark)} \quad [\ frd\ ]$$

Sensing device. The rule *spl* is defined to model the interrupt handler $Sample(dev)$ upon a sampling request, where *dev* is the related sensing component and $Range(dev)$ denotes the data range of the sensing component *dev*. When $\mid Range(dev) \mid > 1$ (i.e., the size of $Range(dev)$ is larger than one), executing $Sample(dev)$ will become non-deterministic.

$$\frac{dt \in Range(dev),\ t_{rd} \notin Q,\ Q' = Q \frown \langle t_{rd} \rangle}{(V,\ Q,\ B,\ Sample(dev)) \overset{s.sensed.dt}{\rightarrow} (V[dt/s.dev.sensed],\ Q',\ B,\ \checkmark)} \quad [\ spl\ ]$$

Concurrent execution of devices. In our work, an interrupt handler is considered as an atomic action. Thus we ignore the preemption of interrupts. Nevertheless, devices such as Timer, Receiver, Reader (Sensor) and so on 'execute' concurrently, because they can trigger interrupts independently. This is captured using an interleave operator $|||$, which resembles the interleave operator in CSP [74]. Rules *int*1 - *int*5 describe the semantics of interleaving. If neither P nor S is termination, then one of them is chosen nondeterministically to execute, as shown in the rules *int*1 and *int*2.

$$\frac{P \neq \checkmark,\ (V,\ Q,\ B,\ P) \overset{e}{\rightarrow} (V',\ Q',\ B',\ P')}{(V,\ Q,\ B,\ P\ |||\ S) \overset{e}{\rightarrow} (V',\ Q',\ B',\ P'\ |||\ S)} \quad [\ int1\ ]$$

$$\frac{S \neq \checkmark,\ (V,\ Q,\ B,\ S) \overset{e}{\rightarrow} (V',\ Q',\ B',\ S')}{(V,\ Q,\ B,\ P\ |||\ S) \overset{e}{\rightarrow} (V',\ Q',\ B',\ P\ |||\ S')} \quad [\ int2\ ]$$

If one of the devices terminates, then the interleave operator is resolved and a non-termination device is executed, as shown in the rules *int*3 and *int*4.

$$\frac{S \neq \checkmark,\ (V,\ Q,\ B,\ S) \overset{e}{\rightarrow} (V',\ Q',\ B',\ S')}{(V,\ Q,\ B,\ \checkmark\ |||\ S) \overset{e}{\rightarrow} (V',\ Q',\ B',\ S')} \quad [\ int3\ ]$$

$$\frac{P \neq \checkmark,\ (V,\ Q,\ B,\ P) \overset{e}{\rightarrow} (V',\ Q',\ B',\ P')}{(V,\ Q,\ B,\ P\ |||\ \checkmark) \overset{e}{\rightarrow} (V',\ Q',\ B',\ P')} \quad [\ int4\ ]$$

If both devices are terminations, then the result is also termination and the interleave

operator is resolved.

$$\frac{}{(V,\ Q,\ B, \checkmark \ |||\ \checkmark) \xrightarrow{\tau} (V,\ Q,\ B,\ \checkmark)} \quad [\ int5\ ]$$

5.2.2 Interrupt Operator

The interrupt operator ($\triangle$) is introduced to formalize the concurrent execution between tasks and interrupts. In TinyOS execution model, interrupts always preempt tasks. When a task terminates, a new task will be fetched from the task queue for execution. A sensor will remain *idle* when no interrupts are triggered by devices and no tasks are scheduled in the task queue. It can be activated again by an interrupt like the arrival of a new message.

If interrupts are enabled (i.e., $V(I) = on$), the hardware devices (H) can run one step.

$$\frac{V(I) = on,\ (V,\ Q,\ B,\ H) \xrightarrow{e} (V',\ Q',\ B',\ H')}{(V,\ Q,\ B,\ M \triangle H) \xrightarrow{e} (V',\ Q',\ B',\ M \triangle H')} \quad [\ itr1\ ]$$

The executing task can perform a step when hardware devices are idle, or interrupts are disabled like when executing an atomic block.

$$\frac{H = idle \text{ or } V(I) = off,\ (V,\ Q,\ B,\ M) \xrightarrow{e} (V',\ Q',\ B',\ M')}{(V,\ Q,\ B,\ M \triangle H) \xrightarrow{e} (V',\ Q',\ B',\ M' \triangle H)} \quad [\ itr2\ ]$$

When a task (M) completes its completion, a new task will be fetched from the task queue (Q) for execution.

$$\frac{H = idle,\ M = Impl(t, \varnothing)}{(V,\ \langle t \rangle^{\frown} Q',\ B,\ \checkmark \triangle H) \xrightarrow{\tau} (V,\ Q',\ B,\ M \triangle H)} \quad [\ itr3\ ]$$

A sensor will remain *idle* when no interrupts are triggered by devices or no tasks are deferred (rule $itr4$), and it can be activated by an interrupt like the arrival of a new message.

$$\frac{H = idle}{(V,\ \varnothing,\ B,\ \checkmark \triangle H) \xrightarrow{s.idle} (V,\ \varnothing,\ B,\ \checkmark \triangle H)} \quad [\ itr4\ ]$$

A sensor will stop if no hardware devices are active and no tasks are pending execution, as in the rule $itr5$.

$$\frac{}{(V,\ \varnothing,\ B,\ \checkmark \triangle \epsilon) \xrightarrow{s.stop} (V,\ \varnothing,\ B,\ Stop)} \quad [\ itr5\ ]$$

5.3 Network Composition

In this section, we formalize SNs as LTSs. A sensor network $\mathcal{N}$ is composed of a set of sensors and a network topology[1]. From a logical point of view, a network topology is simply a directed graph where nodes are sensors and edges are data links between sensors. In reality, a sensor always broadcasts messages and only the ones within its radio range would be able to receive the messages. We introduce a radio range model to describe the network topology, i.e. whether a sensor is able to send messages to some other sensor. Let $\mathbb{N} = \{0, 1, \cdots, i, \cdots, n\}$ be the set of the unique identifier of each sensor in an SN $\mathcal{N}$. The radio range model is defined as the relation $\mathcal{R} : \mathbb{N} \leftrightarrow \mathbb{N}$, such that $(i, j) \in \mathcal{R}$ if and only if sensor j is within sensor i's radio range. We define an SN model as the parallel composition of the sensors with the radio range model, as shown in Definition 11.

Definition 11 (SN Model)**.** *The model of a sensor network $\mathcal{N}$ is defined as a tuple($\mathcal{R}, \{\mathcal{S}_0, \cdots, \mathcal{S}_n\}$) where $\mathcal{R}$ is the radio model (i.e. network topology), $\{\mathcal{S}_0, \cdots, \mathcal{S}_n\}$ is a finite ordered set of sensor models, and $\mathcal{S}_i$ ($0 \leqslant i \leqslant n$) is the model of sensor i.*

For each sensor model $\mathcal{S}_i = (A_i, T_i, R_i, init_i, P_i)$, its initial state is $C_i^{init} = (init_i, \varnothing, \varnothing, P_i)$, in respect that initially task queue T_i and message buffer R_i are empty. Sensors in a network can deliver messages between one another, and semantic structures *Send* and *Rcv* are defined to model message transmission among sensors. SNs are highly concurrent as all sensors run in parallel, i.e. the network behaviors are obtained by non-deterministically choosing one sensor to execute at each step.

Definition 12 (SN State)**.** *Let $\mathcal{N} = (\mathcal{R}, \{\mathcal{S}_0, \cdots, \mathcal{S}_n\})$ be an SN model. A state of $\mathcal{N}$ is defined as the finite ordered set of sensor states: $\mathcal{C} = \{C_0, \cdots, C_n\}$ where C_i ($0 \leqslant i \leqslant n$) is the state of $\mathcal{S}_i$.*

Definition 12 formally defines a global system state of an SN. Based on this, we are now able to represent a sensor network as an LTS.

Definition 13 (SN Transition System)**.** *Let $\mathcal{N} = (\mathcal{R}, \{\mathcal{S}_0, \cdots, \mathcal{S}_n\})$ be an SN model. The transition system corresponding to $\mathcal{N}$ is a 3-tuple $\mathcal{T} = (\Gamma, init, \hookrightarrow)$ where Γ is the set of all reachable SN states, $init = \{C_0^{init}, \cdots, C_n^{init}\}$ (C_i^{init} is the initial state of $\mathcal{S}_i$) being the initial state of $\mathcal{N}$, and $\hookrightarrow$ is the transition relation.*

A transition of an SN is of the form $\mathcal{C} \overset{e}{\hookrightarrow} \mathcal{C}'$, where $\mathcal{C} = \{C_0, \cdots, C_i, \cdots, C_n\}$ and $\mathcal{C}' = \{C_0', \cdots, C_i', \cdots, C_n'\}$. The transition relation is obtained through a set of firing rules, which are shown below.

$$\frac{C_i \overset{e}{\rightarrow} C_i',\ e \neq s_i.send.dst.am.msg,\ e \neq s_i.idle,\ e \neq s_i.stop}{\{C_0, \cdots, C_i, \cdots, C_n\} \overset{e}{\hookrightarrow} \{C_0, \cdots, C_i', \cdots, C_n\}} \quad [\ network1\ ]$$

[1] We assume that the network topology for a given SN is fixed in this work.

Rule *network*1 describes the concurrent execution betweens sensors, i.e. the network non-deterministically chooses a sensor to perform a transition. This rule also indicates that our modeling approach ignores the problem of collision caused by the simultaneous packet transmission of sensor nodes.

$$\frac{C_i \xrightarrow{e} C_i',\ e = s_i.send.dst.am.msg,\ \forall 0 \leq j \leq n, i \neq j \bullet C_j' = InMsg(i, msg, C_j)}{\{C_0, \cdots, C_i, \cdots, C_n\} \overset{e}{\hookrightarrow} \{C_0', \cdots, C_i', \cdots, C_n'\}} \quad [\ network2\ ]$$

Rule *network*2 is dedicated to network communication. Function $InMsg(i, msg, C_j)$ enqueues the message msg to sensor i if it is in the radio range of sensor j. Formally, $C_j' = InMsg(i, msg, C_i) \Leftrightarrow (i,j) \in \mathcal{R} \wedge C_j' = C_j[B_j \frown \langle msg \rangle / B_j] \vee (i,j) \notin \mathcal{R} \wedge C_j' = C_j$. Thus a sensor sending a message will not only change its local state but also may change those of the sensors in its radio range, by enqueuing the message to their message buffers.

As shown in the rule *network*3, the network will remain idle if all sensors are at an idle state.

$$\frac{\forall 0 \leq i \leq n \bullet C_i \overset{s_i.idle}{\rightarrow} C_i}{\{C_0, \cdots, C_i, \cdots, C_n\} \overset{nw.idle}{\hookrightarrow} \{C_0, \cdots, C_i, \cdots, C_n\}} \quad [\ network3\ ]$$

If a sensor stops, the corresponding sensor state is removed from the network state.

$$\frac{C_i \overset{s_i.stop}{\rightarrow} C_i'}{\{C_0, \cdots, C_i, \cdots, C_n\} \overset{s_i.stop}{\hookrightarrow} \{C_0, \cdots, C_{i-1}, C_{i+1}, \cdots, C_n\}} \quad [\ network4\ ]$$

Example 20 (An SN Model)**.** *Figure 5.12 presents an SN with three nodes.*

The radio range of each senor is described by a circle around it. Let $\mathcal{S}_1 = (A_1, T_1, R_1, init_1, P_1)$, $\mathcal{S}_2 = (A_2, T_2, R_2, init_2, P_2)$ and $\mathcal{S}_3 = (A_3, T_3, R_3, init_3, P_3)$ be the sensor models of sensor 1, 2 and 3, respectively. Then the model of the SN is $\mathcal{N} = (\mathcal{R}, \{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3\})$, where $\mathcal{R} = \{(1,2),(2,1),(2,3)\}$. Let $\mathbb{T}_1 = (\mathbb{C}_1, init_1, \rightarrow_1)$, $\mathbb{T}_2 = (\mathbb{C}_2, init_2, \rightarrow_2)$ and $\mathbb{T}_3 = (\mathbb{C}_3, init_3, \rightarrow_3)$ be the sensor transition systems for $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$, respectively. Then the SN transition system is $\mathcal{T} = (\Gamma, init, \hookrightarrow)$ where $\Gamma \subseteq \mathbb{C}_1 \times \mathbb{C}_2 \times \mathbb{C}_3$, and $init = \{C_1^{init}, C_2^{init}, C_3^{init}\} = \{(init_1, \varnothing, \varnothing, P_1), (init_2, \varnothing, \varnothing, P_2), (init_3, \varnothing, \varnothing, P_3), \}$.

5.4 Modeling Environments

In order to model the unreliable environments, we consider the following features.

- Environment data. In our approach, we provide an interface to specify the value range of each sensing device for each sensor, which will be presented in Section 7.2.1. The rule *spl* for data sampling then decides the range of value that can be sensed by the

range specified. The unreliable environment can be modeled by specifying the value range of sensing device. For example, in order to analyze a fire detection system, we can specify the range of value of the temperature sensing device to include normal temperature values and abnormally high temperature values.

- Unreliable communication. In our approach, the message buffer of each sensor is limited and we allow users to specify the size of message buffer of each sensor. When the buffer is full, incoming packets are simply dropped and in this way the behavior of packet loss is modeled in our approach. Packet loss due to unreliable data links is not modeled in our approach, because modeling unreliable data links without probability will always introduce false-negatives of the network being analyzed.

- Uncertain interrupts. On one hand, interrupts are allowed to interleave the execution of a task at each single statement of the task. On the other hand, enabled interrupts are allowed to concurrently execute. Thus, the behavior of interrupts is exhaustively modeled and the uncertainty of interrupt events is resolved.

5.5 Model Checking Algorithms

Algorithm 1 DFS

$DFS(\mathcal{C}_0, \varphi)$

```
1: if C_0 violates φ then
2:    return false
3: end if
4: vs ← {C_0}
5: ws ← {C_0}
6: while ws ≠ ∅ do
7:    C ← ws.Pop()
8:    for all C' such that ∃ α. C ↪^α C' do
9:       if C' violates φ then
10:         return false
11:      end if
12:      if C' ∉ vs then
13:         ws ← ws.Push(C')
14:         vs ← vs ∪ {C'}
15:      end if
16:   end for
17: end while
18: return true
```

Our approach supports the verification of both safety properties and liveness properties, as introduced in Section 2.4. In the following, we introduce the algorithms for verifying these properties.

Safety properties are checked by exhaustively exploring the state space using Depth-first search (DFS) or Breadth-first search (BFS) algorithms. We present the DFS algorithm in 1, and the BFS algorithm is developed in a similar way. As shown in Algorithm 1, two stacks *vs* (visited states) and *ws* (working states) are maintained to keep track of the states that have been visited and the states to be explored for successors, respectively. Both stacks are initialized with the initial state $\mathcal{C}_0$ of the sensor network to be model checked. The

loop between lines 6 and 17 takes a state from *ws* and applies the firing rules introduced in Sections 5.1 - 5.3 to generate subsequent states. The algorithm terminates immediately if a state violating the property φ is encountered (lines 9 and 10). Otherwise, new states that do not exist in *vs* are pushed to *ws* and *vs* (lines 12 to 17). If no states violate φ, the algorithm terminates when *ws* becomes $\varnothing$, which means that the complete state space of the sensor network has been explored.

We adopt the approach presented in [144] to verify LTL properties. First, the negation of an LTL formula is converted into an equivalent Büchi automaton; and then accepting strongly connected components (SCC) in the synchronous product of the automaton and the model are examined in order to find a counterexample. Notice that the SCC-based algorithm allows us to model check with fairness [149], which often plays an important role in proving liveness properties of SNs.

5.6 Experiment and Evaluation

The direct verification approach of SNs has been implemented in the tool NesC@PAT, which will be explained in Chapter 7. In this section, we evaluate the direct verification approach and present experiment results. Section 5.6.1 provides a comparison of the direct verification approach and the translation-based approach NesC2STCSP, an Section 5.6.2 presents a case study on the verification of the real-world sensor network protocol, the Trickle algorithm [88].

5.6.1 Comparison with NesC2STCSP

In this section, we compare the translation-based approach NesC2STCSP presented in Chapter 4 with the direct verification approach presented in this chapter.

First, the direct verification approach supports a larger set of the NesC language features than NesC2STCSP. Table 5.2 demonstrates comparison of the supported language features of both approaches, where **N2S** stands for the NesC2STCSP approach and **Direct** stands for the direct verification approach.

Second, the state space obtained by the direct approach has fewer states than the NesC2STCSP approach. Table 5.3 compares the verification statistics of both approaches. It can be seen that the direct approach has much fewer states for the Sensing application than the NesC2STCSP approach. This confirms that direct verification approaches are more feasible and efficient for the purpose of verification.

5.6.2 The Trickle Algorithm: a Case Study

The direct verification approach has been used to analyze SNs deployed with the Trickle algorithm presented in Example 1. We studied SNs with different topologies including star, ring and single-tracked ring (SRing for short). The settings of SNs are presented in

Category	Features		N2S	Direct
Inherited from C	Pointer		×	✓
	Struct		×	✓
	Data Types	32-bit integer	✓	✓
		Boolean	✓	✓
		Float	×	✓
		Double	×	✓
		8/16/64-bit (un)signed integer	×	✓
	Algebraic operations	$+, -, \times, /, \%$	✓	✓
	Logical operations	$\&\&, \|\|, \neg, =, \neq, <, \leq, >, \geq$	✓	✓
	Bitwise operations	$\&, \|, \sim, \wedge, >>, <<$	×	✓
	Label statement		×	✓
NesC specific	Generic components and interfaces		×	✓
	Fan-in(out) wirings		×	✓
	Atomic statement		×	✓

Table 5.2: Comparison of Language Features Supported

Figure 5.13, where 1/1 stands for *new* code/*new* summary and 0/0 stands for *old* code/*old* summary and a directed graph is used to illustrate the logical view for each network.

Two safety properties (i.e. *Non-termination* and *Reach FalseUpdated*) and two liveness properties (i.e. $\Diamond$*AllUpdated* and $\Box\Diamond$*AllUpdated*) are verified. Property *Reach FalseUpdated* is a state reachability checking, which is valid if and only if the state *FalseUpdated* where a node updates its code with an older one can be reached. Property $\Diamond$*AllUpdated* is an LTL formula which is valid if and only if the state *AllUpdated* where all nodes are updated with the new code can be reached *eventually*, while $\Box\Diamond$*AllUpdated* requires state *AllUpdated* to be reached *infinitely often*.

In order to specify these assertions, we define the following propositions, where n stands for the size of the related network.

- FalseUpdated $\equiv$ S_1.code $<$ S_1.ID || S_2.code $<$ S_2.ID || $\cdots$ || S_{n-1}.code$<$ S_{n-1}.ID;
- AllUpdated $\equiv$ S_1.code $==$ S_n.ID && S_2.code $==$ S_n.ID && $\cdots$ && S_n.code $==$$S_n$.ID.

Based on these propositions, the two safety properties *Non-termination* and *Reach FalseUpdated* are specified as follows.

- *Non-termination*: #assert SensorNetwork never Terminates;
- *Reach FalseUpdated*: #assert SensorNetwork never FalseUpdated.

As for liveness properties $\Diamond$*AllUpdated* and $\Box\Diamond$*AllUpdated*, they are specified as the following.

- $\Diamond$AllUpdated: #assert SensorNetwork |= <>AllUpdated;

App	Assertion	Result	NesC2STCSP		Direct	
			# States	Time (s)	# States	Time (s)
Blink	Non-termination	True	41	0.01	32	0.01
	□◇ TimerFired	True	86	0.01	67	0.01
	□ (TimerFired → ◇ led0Toggled)	True	54	0.01	68	0.01
Blink'	Non-termination	True	243	0.02	85	0.01
	□◇TimerFired	True	706	0.08	199	0.01
	□ (TimerFired → ◇ led0Toggled)	True	417	0.05	202	0.01
Sensing	Non-termination	True	1048K	60	590	0.02
	□◇TimerFired	True	4313K	369	1533	0.13
	□ (TimerFired → ◇ led0Toggled)	True	2598	239	1538	0.15

Table 5.3: Comparison of Performance Features Supported

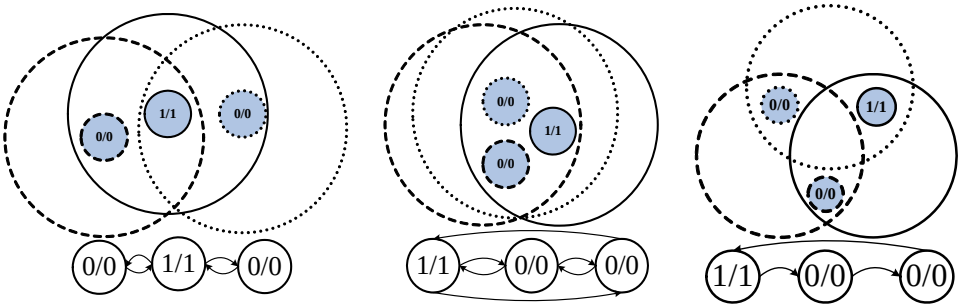


Figure 5.13: Network Topology: Star, Ring, Single-track Ring

- □◇AllUpdated: #assert SensorNetwork |= []<>AllUpdated.

A server with Intel Xeon 4-Core CPU*2 and 32 GB memory was used to run the verification, and the results are summarized in Table 5.4. The results show that the algorithm satisfies the safety properties, i.e. neither a terminating state nor a *FalseUpdated* state is reachable. As for the liveness property, both ring and star networks work well, which means that every node is eventually updated with the new code.

However, the single-tracked-ring (SRing) network fails to satisfy either liveness property. The counterexample produced by NesC@PAT shows that only one sensor can be updated with the new code. By simulating the counterexample in NesC@PAT, we can find the reason for this bug. On one hand, the initially updated node A receives old summary from node C thus broadcasting its code but only node B can hear it. On the other hand, after node B is updated it fails to send its code to node C because node B never hears an older summary from node C. We also increased the number of sensors for SRing networks, and the results remained the same.

(a) Initial Setting (b) Star/Ring network (c) SRing network

Figure 5.14: Real Executions on Iris Motes

Network	Property	Size	Result	#State	#Transition	Time(s)	Memory(KB)
Star	Non-termination	3	✓	300,332	859,115	49	42,936
	Reach FalseUpdated		×	300,332	859,115	47	23,165
	◇AllUpdated		✓	791,419	2,270,243	148	25,133
	□◇AllUpdated		✓	1,620,273	6,885,511	654	13,281,100
Ring	Non-termination	3	✓	1,093,077	3,152,574	171	80,108
	Reach FalseUpdated		×	1,093,077	3,152,574	161	27,871
	◇AllUpdated		✓	2,127,930	6,157,922	389	78,435
SRing	Non-termination	3	✓	30,872	85,143	5	19,968
		4	✓	672,136	2,476,655	170	72,209
	Reach FalseUpdated	3	×	30,872	85,143	5	23,641
		4	×	672,136	2,476,655	156	62,778
	◇AllUpdated	3	×	42	73	<1	19,290
		4	×	52	113	<1	19,771
	□◇AllUpdated	3	×	146	147	<1	51,938
		4	×	226	227	<1	51,421
		8	×	746	747	<1	59,900
		20	×	4,126	4,127	2	148,155

Table 5.4: Verifying Trickle Algorithm

We ran the Trickle program on IRIS motes to study whether this bug could be evidenced in real executions. The *TrickleAppC* was modified by adding operations on LEDs to display the changes of code:

1. Initially sensor A has the new code (1/1) and has its red LED on, while sensor B and C have the old code (0/0) and have their yellow LEDs on, as shown in Figure 5.14(a).

2. A sensor will turns on its red LED when it is updated with the new code.

3. Different topologies are achieved by specifying different *AM id* for each sensor's *AMSenderC* and *AMReceiverC*.

The revised *TrickleAppC* was executed on real sensors with star, ring and single-tracked ring topologies. Figure 5.14(b) shows that the star/ring network is able to update all nodes, and Figure 5.14(c) shows that the single-tracked ring network fails to update one node, which confirms that the bug found by NesC@PAT could be evidenced in reality.

Category	Sum	Construct	Number	Rules
Inherited from C	37	Expression	3	$expr$1-3
		Assignment	2	$as1, 2$
		Function Invocation	2	$inv1, 2$
		Function	2	$func1, 2$
		Label	1	gt
		Return	3	$return$1-3
		Sequence	2	$seq1, 2$
		If-else	3	if1-3
		While/For	17	wh1-6, for1-8, $loop$1-3
		Continue	1	$cont$
		Break	1	brk
NesC specific	11	Atomic	4	atm1-5
		Post	2	$post1, 2$
		Call/Signal	4	$call1, 2;\ sig1, 2$
TinyOS Execution Model	14	Interrupt	5	itr1-5
		Device Operation	4	$send, rcv, spl, frd$
		Device Interleave	5	int1-5
Network composition			4	$network$1-4
Total				66

Table 5.5: Summary of semantic rules

Discussion The results in Table 5.4 show that a SN of the Trickle algorithm with three nodes has a state space of $10^6 \sim 10^7$, and the state space grows exponentially with the network topology and the number of nodes. Thus there is a need for techniques to reduce the state space. Hardware-related behaviors are abstracted and manually modeled based on real sensors, i.e. Iris motes. This manual abstraction of hardware is simple to implement, but lacks the ability to find errors due to hardware failures, as the hardware is assumed to be always working.

5.7 Summary

In this chapter, we present the complete semantics and define firing rules for the language constructs of NesC. The complete semantics for sensor networks consists of 66 firing rules, as summarized in Table 5.5. With this formal semantics, it is now possible to build the system state space directly from sensor network implementations directly.

Verification goals are presented in a lightweight annotation language separated from the NesC language or TinyOS. In this way, assertions could be defined without modifying the original sensor network programs, and thus it makes it convenient to put the same copy of code being model checked into practice. The approach has been implemented in the tool NesC@PAT, which will be discussed in Chapter 7. We have studied the performance

of this approach with a number of examples, and results show that our approach helps to analyze and verify SNs with NesC programs effectively. The comparison between the direct verification approach and the translation-based approach NesC2STCSP confirms that direct verification approach works better.

The main challenge of this approach (and general model checking methods) is the state space explosion problem. The concurrency among sensors and among interrupts inside each sensor has introduced a huge state space which is usually infeasible to be completely explored within acceptable time and available memory. Therefore, there is a need to develop techniques to reduce the system state space without losing the ability to analyze and check the system. In the next chapter, we will discuss a novel two-level partial order reduction method specifically for sensor network systems.

Chapter 6

Reduction and Optimization

In this chapter, we discuss a method developed 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. The 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 [65] 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 [34].

This chapter is organized as follows. In Section 6.1, we present the theoretical techniques for analyzing the independence relation among actions in SNs. In Section 6.2, we present the sensor network cartesian semantics, which is based on the cartesian semantics for concurrent programs [65]. And then in Section 6.3, we present our novel two-level POR for SNs, which extends the POR in [65] to allow reduction at different system levels. In Section 6.4, we formally prove that our method is sound and complete, i.e., preserving LTL-X properties [34]. The proposed method has been implemented in the model checker NesC@PAT [172] 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 approximate the reduction ratio obtained by a related tool T-Check [89] 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 6.5. A summary of this chapter is presented in Section 6.6.

In this chapter, the concepts of independence, equivalence and so on are discussed w.r.t. a given property φ.

6.1 Static Analysis

In this section, we study the commutativity of actions in a sensor network. In this thesis, two actions are said to be independent if and only if their execution order is commutative. In particular, we examine the independence relation between actions within an arbitrary sensor, and between actions from different sensors. These are important as in SNs, there are inter-sensor concurrency and intra-sensor concurrency. The independence analysis rules are elaborated and we propose a systematic way to detect independence among interrupts by taking into account the shared task queue. These will be used for automatic identification of independent actions for the two-level POR approach which will be covered later.

For a sensor network, $\mathcal{C}$ denotes a network state, defined as $\{C_0, \cdots, C_n\}$ where C_i $(0 \leqslant i \leqslant n)$ is the state of $\mathcal{S}_i$, also denoted by $\mathcal{C}[i]$. In the rest of this chapter, the following definitions and notations are used.

- $enable(C)$: the set of all actions enabled at a sensor state C, i.e., $enable(C) = \{\alpha \mid \exists\, C' \in \mathbb{C}, C \xrightarrow{\alpha} C'\}$.
- $ex(C, \alpha)$ (where $\alpha \in enable(C)$): the sensor state after executing α at state C.
- $\sum_{\mathcal{S}}$ (or simply $\sum$ if $\mathcal{S}$ is clear): the set of actions of $\mathcal{S}$. We assume that $\sum$ is finite.
- $\sum^{iq} \subseteq \sum$: the set of hardware request actions.
- $\sum^{pt} \subseteq \sum$: the set of actions that are *post* statements.
- $itrH(\mathcal{S}) \subseteq \sum$: the set of interrupt handlers.
- $\sum^{sd} \subseteq \sum$: the set of actions involving packet transmission.
- $\sum^{asyn} \subseteq \sum$: the set of actions executed in an asynchronous context, i.e., in an interrupt handler.
- $\sum^{syn} \subseteq \sum$: the set of actions executed in a synchronous context, i.e., in a normal task.
- $Tasks(\mathcal{S})$ (or simply $Tasks$ if $\mathcal{S}$ is clear): the set of all tasks defined in $\mathcal{S}$. We assume that $Tasks$ is finite.

6.1.1 A Motivating Example

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 Figure 5.10(b) presented in Section 5.2, 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,

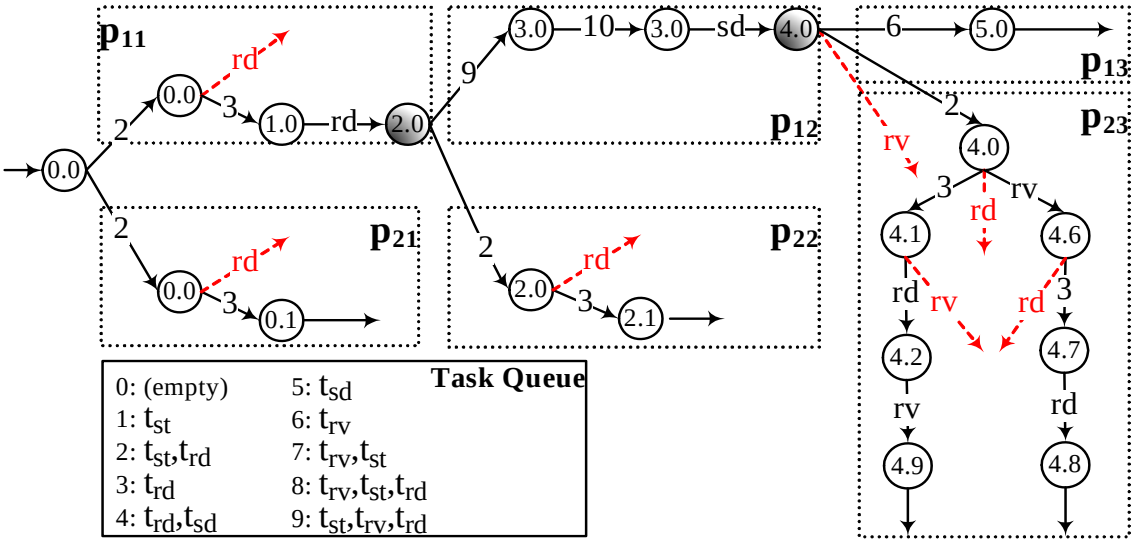


Figure 6.1: Pruned State Graph

like *rd* and *rv* at state 2, and line 3 and *rd* at state 1. This is because t_{rd} only accesses the 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 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 Figure 5.10(a). Applying partial order reduction at both network and sensor levels, we are able to obtain a reduced state graph as shown in Figure 6.1, 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 $\mathcal{S}_2$ where there is communication between the two sensors. 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 Figure 5.10(b). It can be seen that the corresponding complete state space of this graph consists of around 200 states, whereas the reduced graph contains fewer than 20 states.

In the following, we investigate local and global independence w.r.t. a certain property, since the independence relation of actions or tasks would be different if different properties are considered.

6.1.2 Local Independence

In a sensor, an action may modify a variable or the task queue. Thus local independence is defined to describe independent actions according to their 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 model $\mathcal{S}$.

Definition 14 (Variable 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 variable-independent, denoted by $\alpha_1 \equiv_{VI} \alpha_2$, iff $ex(ex(C, \alpha_1), \alpha_2) =_v ex(ex(C, \alpha_2), \alpha_1)$.*

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 $V_1 = V_2 \wedge B_1 = B_2 \wedge P_1 = P_2$. Let W_α and R_α be the set of variables written and only read by an action α, respectively. Let $C(a)$ be the valuation function of variable a at state C.

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 $ex(C_0, \alpha_1) = C_{01}$, $ex(C_{01}, \alpha_2) = C_{12}$, $ex(C_0, \alpha_2) = C_{02}$, and $ex(C_{02}, \alpha_1) = C_{21}$. Let V_0, V_{01}, V_{12}, V_{02} and V_{21} be the valuation of C_0, C_{01}, C_{12}, C_{02} and C_{21}, respectively. Since $W_{\alpha_2} \cap (W_{\alpha_1} \cup R_{\alpha_1}) = \varnothing$, we can conclude that $\forall\, x \in (W_{\alpha_1} \cup R_{\alpha_1})$. $V_{02}(x) = V_0(x) \wedge V_{01}(x) = V_{12}(x)$, further $V_{21}(x) = V_{01}(x)$, and finally $V_{12}(x) = V_{21}(x)$. Similarly, we can prove that $\forall\, x \in (W_{\alpha_2} \cup R_{\alpha_2})$. $V_{21}(x) = V_{12}(x)$. Let Var be the set of all variables and we can conclude immediately that $\forall\, x \in (Var - (R_{\alpha_1} \cup W_{\alpha_1} \cup R_{\alpha_2} \cup W_{\alpha_2}))$. $V_{12}(x) = V_{21}(x)$. Therefore, $V_{12} = V_{21}$ and $C_{12} =_v C_{21}$. □

Lemma 1 shows that two actions are variable-independent if the variables modified by one action are mutual exclusive with those accessed (either modified or read) 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 (13) of Figure 5.10(a).

Interrupt handlers might run in parallel resulting in 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, $Ptask(t) = \{t' \mid \exists\, \alpha \in t.\ \alpha = post(t') \vee (\alpha \in \sum^{iq}) \wedge t' = tsk(ih(\alpha))\}$, 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 Figure 5.10(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 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 Figure 5.10(a), 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 $Rtask(t)$. We say that t is a φ-safe task, denoted by $t \in safe(\varphi, \mathcal{S})$ iff $(\widehat{W}(t) \cap R(\varphi, \mathcal{S})) = \varnothing$. In the following, we define local independence of tasks.

Definition 15 (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 $(t_i \in safe(\varphi, \mathcal{S}) \vee t_j \in safe(\varphi, \mathcal{S})) \ \wedge \ \forall\, t_i' \in Rtask(t_i), t_j' \in Rtask(t_j).\ \forall\, \alpha_i \in t_i', \alpha_j \in t_j'.\ \alpha_i \equiv_{VI} \alpha_j$.*

In the above definition, the condition that one of the task should be φ-safe guarantees that executing the two tasks in different orders will introduce no difference on the effect of the property φ.

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. Based on Definition 15, we define the independence relation of two task sequences as follows, which is used to further define equivalent task sequences.

Definition 16 (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 $\forall\, t_i \in (\cup_{k=0}^{m} Rtask(t_{ik})), t_j \in (\cup_{k=0}^{n} Rtask(t_{jk}))$. $t_i \equiv_{TI} t_j$.*

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 $\mathcal{P}$, $Swap^{\mathcal{P}}(Q, i) = q_0^\frown \cdots q_{i+1}^\frown q_i^\frown \cdots^\frown q_n$ denotes the task sequence obtained by swapping two adjacent sub-sequences (i.e., q_i and q_{i+1}) consistent with $\mathcal{P}$, and $Q = \langle q_0^\frown q_1^\frown \cdots^\frown q_n\rangle$. Then task sequence equivalence is defined in Definition 17, which is adopted to reduce unnecessary interleaving among interrupt handlers and actions.

Definition 17 (Task Sequence Equivalence)**.** *Given two task sequences Q and Q', they are equivalent ($Q \simeq Q'$) iff $Q^0 = Q \ \wedge \ \exists \mathcal{P}.\ \exists\, m \geqslant 0, Q^m = Q' \ \wedge \ (\forall\, k \in [0, m).\ \exists\, i_k.\ q_{i_k}^k \equiv_{SI} q_{i_k+1}^k \ \wedge \ Q^{k+1} = Swap^{\mathcal{P}}(Q^k, i_k))$ where q_i^k 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 $Q \simeq Q'$. 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 17, $m = 0 \Rightarrow Q_1^m = Q_1^0 = Q$, 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 17, we have $Q_1^0 = Q_1 \Rightarrow \exists\, m > 0, Q_1^m = Q_2 \wedge (\forall\, 1 \leq k \leq m, \exists\, i_k, q_{i_k}^{k-1} \equiv_{SI} q_{i_k+1}^{k-1} \wedge Q_1^k = Swap^{\mathcal{P}}(Q_1^{k-1}, i_k))$ (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 $Q_2^k = Swap^{\mathcal{P}}(Q_2^{k-1}, i_{m-(k-1)})$. Then we have $Q_2^0 = Swap^{\mathcal{P}}(Q_1^{m-1}, i_m) \wedge \widetilde{q}_{i_m}^0 = q_{i_m+1}^{m-1} \wedge \widetilde{q}_{i_m+1}^0 = q_{i_m}^{m-1}$, thus $\widetilde{q}_{i_m}^0 \equiv_{SI} \wedge \widetilde{q}_{i_m+1}^0$ and $Q_2^1 = Swap^{\mathcal{P}}(Q_2^0, i_m) = Swap^{\mathcal{P}}(Swap^{\mathcal{P}}(Q_1^{m-1}, i_m), i_m) = Q_1^{m-1}$. Suppose that $Q_2^k = Q_1^{m-k} = Swap^{\mathcal{P}}(Q_1^{m-k-1}, i_{m-k})$, 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 $Q_2^{k+1} = Swap^{\mathcal{P}}(Q_2^k, i_{m-k}) = Swap^{\mathcal{P}}(Q_1^{m-k}, i_{m-k}) = Swap^{\mathcal{P}}(Swap^{\mathcal{P}}(Q_1^{m-(k+1)}, i_{m-((k+1)-1)}, i_{m-k}) = Q_1^{m-(k+1)}$. 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 17, there exist $i_1, i_2, \cdots, i_m$ such that $Q_1^0 = Q_1 \wedge q_{i_k}^{k-1} \equiv_{SI} q_{i_k+1}^{k-1} \wedge Q_1^k = Swap^{\mathcal{P}}(Q_1^{k-1}, i_k) \wedge Q_1^m = Q_2$ and $j_1, j_2, \cdots, j_n$ such that $Q_2^0 = Q_2 \wedge q_{j_k}^{k-1} \equiv_{SI} q_{j_k+1}^{k-1} \wedge Q_2^k = Swap^{\mathcal{P}}(Q_2^{k-1}, j_k) \wedge Q_2^n = Q_3$. Again, let $\widetilde{Q}_1^0 = Q_1$, $v = m + n$, and $\forall\, 1 \leq k \leq v$, let $\widetilde{Q}_1^k = Swap^{\mathcal{P}}(\widetilde{Q}_1^{k-1}, l_k)$ 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. □

Then we define two actions to be locally independent as the following definition, based on their effects on the variable evaluation and the task queue.

Definition 18 (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 two conditions are satisfied.*

1. $\alpha_1 \equiv_{VI} \alpha_2$;
2. $Q(ex(ex(C, \alpha_1), \alpha_2)) \simeq Q(ex(ex(C, \alpha_2)), \alpha_1)$.

Now we are able to conclude that a non-post action in the synchronous context is always local-independent with any action in the asynchronous context, as shown in Lemma 3.

Lemma 3. $\forall\, \alpha \in \sum^{syn}, \alpha' \in \sum^{asyn} .\ \alpha \notin \sum^{pt} \Rightarrow \alpha \equiv_{LI} \alpha'$.

Proof From $\alpha' \in \sum^{asyn}$, we can conclude that α' only accesses device variables which cannot be accessed by synchronous actions including α. Thus we are able to conclude that $\alpha \equiv_{VI} \alpha'$. Suppose that $\alpha, \alpha \in enable(C)$. Since $\alpha \notin \sum^{pt}$ implying executing α never changes the task queue, we can obtain $Q(ex(C, \alpha)) = Q(C)$. Therefore, $Q(ex(ex(C, \alpha), \alpha')) = Q(ex(ex(C, \alpha), \alpha'))$, and by Lemma 2 we have $Q(ex(ex(C, \alpha), \alpha')) \simeq Q(ex(ex(C, \alpha), \alpha'))$. Now we can obtain that $\alpha \equiv_{LI} \alpha'$ according to Definition 18. □

The following lemma shows another property of task sequence equivalence ($\simeq$).

Lemma 4. $Q_1 \simeq Q_2 \Rightarrow Q_1^{\frown} Q' \simeq Q_2^{\frown} Q' \wedge Q'^{\frown} Q_1 \simeq Q'^{\frown} Q_2$.

Proof By Definition 17, there exists $m \geq 0$ such that $Q_0 = Q_1 \wedge Q^m = Q_2 \wedge \exists\, i_0, i_1, \cdots, i_k, \cdots, i_{m-1}.\ q^k_{i_k} \equiv_{SI} q^k_{i_k+1} \wedge Q^{k+1} = Swap^{\mathcal{P}}(Q^k, i_k)$. Suppose that $Q_3 = Q_1^{\frown} Q'$ and $Q_4 = Q_2^{\frown} Q'$, then $Q_0 = Q_3 \wedge Q^m = Q_4 \wedge \forall\, i_0, i_1, \cdots, i_k, \cdots, i_{m-1}.\ q^k_{i_k} \equiv_{SI} q^k_{i_k+1} \wedge Q^{k+1} = Swap^{\mathcal{P}}(Q^k, i_k)$. Thus $Q_3 \simeq Q_4$. Similarly, we can prove $Q'^{\frown} Q_1 \simeq Q'^{\frown} Q_2$. □

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 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 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 state which is the completion point of a previous task, i.e., a state with the program as $(\checkmark \triangle H)$. This is because 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 6.1.3.

Let $exs(Q, C)$ be the set of final states after executing all tasks of Q starting at state C, which requires that the task queue of state C is exactly Q. In our work, actions are deterministic. During the execution of a task (which is a sequence of actions), interrupts are concurrently executed with the task if interrupts are enabled. Thus the execution of the task queue $exs(Q, C)$ implicitly includes the concurrent execution with interrupt handlers.

Lemma 5. *Given* $C = (V, Q, \langle\rangle, \checkmark \triangle H)$ *and* $C' = (V, Q', \langle\rangle, \checkmark \triangle H)$. $Q \simeq Q' \Rightarrow exs(Q, C) \asymp exs(Q', C')$.

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 17, we have $Q^m = Q'$ and for each k there exists i_k such that $q^k_{i_k} \equiv_{SI} q^k_{i_k+1} \wedge Q^{k+1} = Swap^{\mathcal{P}}(Q^k, i_k))$. 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 16 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 $\forall 1 \leq u \leq x, \widetilde{t}^u_{i_k} \in \cup_{v=1}^{l} Ptask(t^v_{i_k})$ and $\forall 1 \leq u \leq y, \widetilde{t}^u_{i_k+1} \in \cup_{v=1}^{l} Ptask(t^v_{i_k+1})$ (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 $Rtask$, $\cup_{u=1}^{x} Rtask(\widetilde{t}^u_{i_k}) \subseteq \cup_{u=1}^{i} Rtask(t^u_{i_k})$ and $\cup_{u=1}^{y} Rtask(\widetilde{t}^u_{i_k+1}) \subseteq \cup_{u=1}^{l} Rtask(t^u_{i_k+1})$. 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 5 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 task that could be enqueued by executing α. We remark that executing α at most enqueues one task to the task queue, but sometimes which task to be enqueued is only decided at runtime, e.g., $if(sum > 10)\{post\ tsk_1\} else\{post\ tsk_2\}$. With the above lemma, the rule for deciding local independence between actions can be obtained, as shown in Lemma 6.

Lemma 6. *Given* C, $\alpha_1, \alpha_2 \in enable(C)$, $(\alpha_1 \equiv_{VI} \alpha_2 \wedge \forall\, t_1 \in ptsk(\alpha_1), t_2 \in ptsk(\alpha_2).\ t_1 \equiv_{TI} t_2) \Rightarrow \alpha_1 \equiv_{LI} \alpha_2$.

Proof Since $\alpha_1 \equiv_{VI} \alpha_2$, we only need to prove the second condition in Definition 18. Suppose $Q(C) = Q_0$, $ex(ex(C, \alpha_1), \alpha_2) = C_{12}$, $ex(ex(C, \alpha_2), \alpha_1) = C_{21}$ and $Q(C_{12}) = Q_0^{\frown} tsk_1^{\frown} tsk_2$. Since $\alpha_1 \equiv_{VI} \alpha_2$, then we can have $Q(C_{12}) = Q_0^{\frown} tsk_2^{\frown} tsk_1$. Since $\forall\, t_1 \in ptsk(\alpha_1), t_2 \in ptsk(\alpha_2).\ t_1 \equiv_{TI} t_2$, we can conclude that $tsk_1^{\frown} tsk_2 \simeq tsk_2^{\frown} tsk_1$. Therefore, $Q(C_{12}) \simeq Q(C_{21})$. □

6.1.3 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 network 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 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 19 (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 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' tasks even if there is some 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.

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

Proof Supposing that $t_2 \notin \mathit{Sends}(\mathcal{S}_j)$, since $t_1 \notin \mathit{Sends}(\mathcal{S}_i)$, we can prove immediately that $t_1 \equiv_{GI} t_2$. Suppose that $t_2 \in \mathit{Sends}(\mathcal{S}_j)$ 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 $t_1, t_2 \in \mathit{EnableT}(\mathcal{C}_0)$ and $\mathcal{C}_0[i] = (V, \langle t_1\rangle^{\frown} q_i)$, and let $\mathcal{C}_2 \in \mathit{Ex}(\mathcal{C}_0, t_2)$ and $\mathcal{C}_{21} \in \mathit{Ex}(\mathcal{C}_2, t_1)$. Let $\mathcal{C}_{21}[i] = (V^i_{12}, q_i^{\frown}\langle t'_1 \cdots t_{rv} \cdots t'_m\rangle, \varnothing, \checkmark \triangle H)$, 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 $t \in \mathit{Ptask}(t_1)$. It is immediately that there exists $\mathcal{C}_1 \in \mathit{Ex}(\mathcal{C}_0, t_1)$ such that there exists $\mathcal{C}_{12} \in \mathit{Ex}(\mathcal{C}_1, t_2)$, $\mathcal{C}_{12}[j] = \mathcal{C}_{21}[j]$, $\mathcal{C}_{12}[i] =_v \mathcal{C}_{21}[i]$ 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 5, 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. □

6.2 Cartesian Semantics

In this section, we present our two-level POR, which extends the cartesian vector approach [65] and combines it with a persistent set algorithm [61] to achieve maximum reduction. We use $\mathcal{N}$ to denote a sensor network with n sensors $(\mathcal{S}_1, \cdots, \mathcal{S}_n)$; and $\mathcal{C}$ to denote a network state of $\mathcal{N}$.

The cartesian vector approach, also referred to as Cartesian POR, was proposed by Gueta et al. to reduce nondeterminism in concurrent systems [65], which delays unnecessary context switches among processes. 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 [65]. 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 the 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 by $\alpha \in \mathit{safe}(\varphi)$, 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 $\alpha \notin \mathit{safe}(\varphi)$, then α is said to be φ-*unsafe*.

6.2.1 Sensor Prefix

In order to allow sensor-level nondeterminism inside prefixes, we define *Prefix* as a tree-like trace 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 a network state $\mathcal{C}$, and $\mathit{first}(p)$ be the initial state of a prefix p. A sensor prefix is defined as follows.

Definition 20 (Prefix)**.** *A prefix $p \in \mathit{Prefix}(\mathcal{S})$ is defined as a tuple $(\langle \mathcal{C}_0, \alpha_1, \mathcal{C}_1, \cdots, \alpha_{m-1}, \mathcal{C}_m\rangle, \{br_0, br_1, \cdots, br_n\})$, where $\forall\, 1 \le i < m.\ \alpha_i \in \sum_{\mathcal{S}} \ \wedge\ \mathcal{C}_i \overset{\alpha_i}{\hookrightarrow} \mathcal{C}_{i+1}$, and $\forall\, 0 \le i \le n. br_i \in \mathit{Prefix}(\mathcal{S}, \mathcal{C}_m)$.*

Here $\mathit{Prefix}(\mathcal{S},\mathcal{C})$ denotes the set of prefixes of sensor $\mathcal{S}$ starting at state $\mathcal{C}$. The function $tr(p)$ denotes the trunk prefix of p before branching prefixes, i.e., $tr(p) = \langle \mathcal{C}_0, \alpha_1, \mathcal{C}_1, \cdots, \alpha_{m-1}, \mathcal{C}_m\rangle$. The function $br(p)$ denotes the set of branching prefixes of p, i.e., $br(p) = \{br_0, br_1, \cdots, br_n\}$. In Figure 6.1, 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 $br(p_{23}) = \{\langle (4.0), \alpha_3, (4.1), \alpha_{rd}, \cdots\rangle, \langle (4.0), \alpha_{rv}, (4.6), \alpha_3, \cdots\rangle\}$.

If there are no branching prefixes, then the prefix is also said to be a *leaf* prefix, denoted by $p \in \mathit{LfPrefix}(\mathcal{S})$. By Definition 20, $tr(p) \in \mathit{LfPrefix}(\mathcal{S})$. Moreover, it is obvious that $\mathit{LfPrefix}(\mathcal{S}) \subseteq \mathit{Prefix}(\mathcal{S})$. Formally, we have $\mathit{LfPrefix}(\mathcal{S}) = \{lp \mid \forall\, lp \in \mathit{Prefix}(\mathcal{S}).\ br(lp) = \varnothing\}$. A leaf prefix always appears as $(\langle \mathcal{C}_0, \alpha_1, \mathcal{C}_1, \cdots, \alpha_{m-1}, \mathcal{C}_m\rangle, \varnothing)$, and thus is abbreviated as $\langle \mathcal{C}_0, \alpha_1, \mathcal{C}_1, \cdots, \alpha_{m-1}, \mathcal{C}_m\rangle$.

Given a prefix $p \in \mathit{Prefix}(\mathcal{S})$, the following notations are defined, which are later used in the POR algorithms.

- The set of *tree* prefixes of $\mathcal{S}$: $\mathit{TreePrefix}(\mathcal{S}) = \mathit{Prefix}(\mathcal{S}) - \mathit{LfPrefix}(\mathcal{S})$.
- The set of leaf prefixes of p: $p \in \mathit{LfPrefix}(\mathcal{S}) \wedge \mathit{leaf}(p) = \{p\} \ \vee\ p \notin \mathit{LfPrefix}(\mathcal{S}) \wedge \mathit{leaf}(p) = \cup_{bp \in br(p)} \mathit{leaf}(bp)$.
- The final state of a leaf prefix lp: $lp = \langle \mathcal{C}_0, \alpha_0, \mathcal{C}_1, \cdots, \mathcal{C}_m\rangle \Rightarrow \widehat{\mathit{last}}(lp) = \mathcal{C}_m$.
- Subsequent prefixes $\sqsupset$: $\forall\, lp \in \mathit{LfPrefix}(\mathcal{S}).\ lp \sqsupset p \equiv \widehat{\mathit{last}}(lp) = \mathit{first}(p)$.
- Concatenation of leaf prefixes $\widehat{\ }$: $\forall\, p1 = \langle \mathcal{C}_0, \alpha_0, \cdots, \mathcal{C}_k\rangle, p2 = \langle \mathcal{C}_k, \alpha_k, \mathcal{C}_{k_0}, \alpha_{k_0}, \cdots, \mathcal{C}_{k_m}\rangle . p1 \widehat{\ } p2 = \langle \mathcal{C}_0, \alpha_0, \cdots, \mathcal{C}_k, \alpha_k, \mathcal{C}_{k_0}, \cdots, \mathcal{C}_{k_m}\rangle$.
- The set of final states of p: $p \in \mathit{LfPrefix}(\mathcal{S}) \wedge \mathit{last}(p) = \{\widehat{\mathit{last}}(p)\} \ \vee\ p \notin \mathit{LfPrefix}(\mathcal{S}) \wedge \mathit{last}(p) = \cup_{bp \in br(p)} \mathit{last}(bp)$.
- The final task of a leaf prefix lp: $lp = \langle \mathcal{C}_0, \alpha_0, \mathcal{C}_1, \cdots, \mathcal{C}_m\rangle \Rightarrow \widehat{\mathit{lastT}}(lp) = \mathit{currentT}(\mathcal{C}_m)$, where $\mathit{currentT}(\mathcal{C})$ is the currently executing task at state $\mathcal{C}$.

- The set of final tasks of p: $p \in \mathit{LfPrefix}(\mathcal{S}) \wedge \mathit{lastT}(p) = \{\widehat{\mathit{lastT}}(p)\} \vee \mathit{lastT}(p) = \cup_{bp \in br(p)} \mathit{lastT}(bp)$.
- The final action of a leaf prefix lp: $lp = \langle \mathcal{C}_0, \alpha_0, \cdots, \alpha_m, \mathcal{C}_{m+1} \rangle \Rightarrow \widehat{\mathit{lastAct}}(lp) = \alpha_m$.
- The set of final actions of p: $p \in \mathit{LfPrefix}(\mathcal{S}) \wedge \mathit{lastAct}(p) = \{\widehat{\mathit{lastAct}}(p)\} \vee p \notin \mathit{LfPrefix}(\mathcal{S}) \wedge \mathit{lastAct}(p) = \cup_{bp \in br(p)} \mathit{lastAct}(bp)$.
- The set of states in p: $\mathit{states}(p) = \{\mathcal{C}_0, \cdots, \mathcal{C}_m\} \cup (\cup_{sp \in br(p)} \mathit{states}(sp))$.
- The set of tasks of p: $\mathit{tasks}(p) = (\cup_{0 \le i \le m} \mathit{currentT}(\mathcal{C}_i)) \cup (\cup_{sp \in br(p)} \mathit{tasks}(sp))$.
- The set of actions of a leaf prefix lp: $lp = \langle \mathcal{C}_0, \alpha_0, \mathcal{C}_1, \alpha_1, \cdots, \alpha_m, \mathcal{C}_{m+1} \rangle \Rightarrow \mathit{acts}(lp) = \{\alpha_0, \alpha_1, \cdots, \alpha_m\}$.
- The set of actions of a prefix p: $\mathit{acts}(p) = \mathit{acts}(\mathit{tr}(p)) \cup (\cup_{bp \in br(p)} \mathit{acts}(bp))$.

Lemma 7. $\forall\, p_1, p_2 \in \mathit{Prefix}(\mathcal{S}).\ \mathit{tr}(p_1) = \mathit{tr}(p_2) \Rightarrow \mathit{first}(p_1) = \mathit{first}(p_2)$.
Proof This is trivial by the definition of $\mathit{first}(p)$. □

Lemma 8. $\forall\, lp \in \mathit{LfPrefix}(\mathcal{S}), p \in \mathit{Prefix}(\mathcal{S}).\ p' = (lp, \{p\}) \Rightarrow \mathit{first}(lp) = \mathit{first}(p')$.
Proof Since $p' = (lp, \{p\})$, it can be immediately concluded that $\mathit{tr}(p') = \mathit{tr}(lp)$. By Lemma 7, we can obtain $\mathit{first}(lp) = \mathit{first}(p')$. □

6.2.2 SN Cartesian Vector

Definition 21 (SN Cartesian Vector)**.** *Given a global property $\varphi \in \mathit{Gprop}$, a vector $(p_1, \cdots, p_i, \cdots, p_n)$ is a sensor network cartesian vector for $\mathcal{N}$ w.r.t. φ from a network state $\mathcal{C}$* if the following conditions hold*:*

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

According to Definition 21, a vector $(p_0, p_1, \cdots, p_n)$ from $\mathcal{C}$ is a valid sensor network cartesian vector (SNCV) if for every $0 \le i \le n$, 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 6.1.3. In Figure 6.1, if $\mathit{value}, \mathit{busy} \notin R(\varphi)$, 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 a 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 $\Rightarrow_\phi$.

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

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

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

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

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

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

6.3 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. Reduction is performed during the generation of SNCVs. Section 6.3.1 presents the top-level state exploration algorithm, which could be invoked in existing verification algorithms directly without changing the verification engine. Section 6.3.2 presents the algorithm for SNCV generation and Section 6.3.3 discusses the algorithm for generating a prefix for a certain sensor.

6.3.1 State Space Exploration

Given a network state $\mathcal{C}$ and a global property φ, the state space of $\mathcal{N}$ is explored via $\mathcal{C}'s$ corresponding SNCV, as shown in Algorithm 2. The following definitions are used in this algorithm.

- Method $pfx(\mathcal{C})$: returns the prefix that contains $\mathcal{C}$. For the initial state $\mathcal{C}_0$, we have

Algorithm 2 State Space Generation

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

```
1:  list ← ∅
2:  p ← pfx(C)
3:  if Next(p, C) ≠ ∅ then
4:      {there are some successors of C in p}
5:      list ← Next(p, C)
6:  else
7:      {there are no successors of C in p}
8:      {generate a new sncv from C}
9:      scv ← GetNewCV(C, φ)
10:     for all i ← 1 to n do
11:         {traverse each sensor prefix to obtain the
            successors of C}
12:         list ← list ∪ {Next(scv[i], C)}
13:     end for
14: end if
15: return list
```

$pfx(\mathcal{C}_0) = \langle \mathcal{C}_0 \rangle$. For each state $\mathcal{C}'$ further generated, $pfx(\mathcal{C}')$ is assigned during the extension of the corresponding sensor prefix, which can be found in Algorithms 5 and 6.

- Method $Next(p, \mathcal{C})$: returns the successors of $\mathcal{C}$ in p. The relation $Next : Prefix(\mathcal{S}) \times \Gamma \to \mathbb{P}(\Gamma)$ traverses a prefix to find the set of successors of $\mathcal{C}$. Formally, $Next(p, \mathcal{C}) = \{\mathcal{C}' \mid \exists\, \alpha \in acts(p), \mathcal{C} \overset{\alpha}{\hookrightarrow} \mathcal{C}'\}$.
- Method $GetNewCV(\mathcal{C}, \varphi)$: generates a new SNCV starting from $\mathcal{C}$, which is shown in Algorithm 3.

In summary, the responsibility of Algorithm 2 is to look up the prefix of $\mathcal{C}$ to find its successors (lines 3 to 5). If $\mathcal{C}$ is found to be one of the final states of the corresponding prefix, then the algorithm will generate a new SNCV starting from $\mathcal{C}$ (lines 9), and continue to traverse each prefix of the SNCV for the successors of $\mathcal{C}$ (lines 10 to 13).

6.3.2 SNCV Generation

Algorithm 3 generates an SNCV from $\mathcal{C}$ w.r.t. the property φ. In this algorithm, the following notations are used.

- Variable *visited*: the set of final states of prefixes that have been generated.
- Variable *workingLeaf*: the stack of leaf prefixes to be further extended.
- Function *Extensible*: $Prefix(\mathcal{S}) \times \{\mathcal{S}_1, \cdots, \mathcal{S}_n\} \times Gprop \to \{True, False\}$, defined as $Extensible(p, \mathcal{S}, \varphi) \equiv \forall\, t \in lastT(p), \alpha \in lastAct(p).\ t \subset_{GI} \mathcal{S} \ \wedge\ \alpha \in safe(\varphi)$.
- Method $GetPrefix(\mathcal{S}_i, \mathcal{C}, \varphi)$: producing a prefix of $\mathcal{S}_i$ by executing actions and interrupt handlers of $\mathcal{S}_i$ in parallel, which will be shown in Algorithm 4.

Algorithm 3 Sensor Network Cartesian Vector Generation

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

```
1:  scv ← (⟨⟩, ··· , ⟨⟩)   (n)
2:  for all S_i ∈ 𝒩 do
3:     {generate sensor prefix for each sensor}
4:     visited ← {𝒞}
5:     workingLeaf ← ∅
6:     {generate a prefix from 𝒮}
7:     p_i ← GetPrefix(𝒮_i, 𝒞, φ)
8:     for all lp ∈ leaf(p_i) do
9:        {cache extensible leaf prefixes
          to be further extended}
10:       if last(lp) ∉ visited
          and Extensible(lp, 𝒮_i, φ) then
11:          workingLeaf.Push(lp)
12:          visited ← visited ∪ last(lp)
13:       end if
14:    end for
15:    while workingLeaf ≠ ∅ do
16:       p_k ← workingLeaf.Pop()
17:       visited ← visited ∪ {last(p_k)}
18:       {generate a new prefix from the
          last state of p_k}
19:       p'_k ← GetPrefix(𝒮_i, last(p_k), φ)
20:       p_k ← (p_k, {p'_k})
21:       for all lp ∈ leaf(p'_k) do
22:          {cache extensible leaf prefixes
             to be further extended}
23:          if Extensible(lp, 𝒮_i, φ) and
             last(lp) ∉ visited then
24:             workingLeaf.Push(lp)
25:          end if
26:       end for
27:    end while
28:    {update the ith element of scv with p_i}
29:    scv[i] ← p_i
30: end for
31: return scv
```

In order to generate an SNCV, the algorithm executes each sensor in the network independently to obtain the corresponding sensor prefixes that form the SNCV. In order to minimize inter-sensor interleaving, a prefix is extended as much as possible. Initially, *GetPrefix*() is invoked to obtain a sensor prefix p_i (line 7). Then each leaf prefix of p_i are examined and pushed into *workingLeaf* stack if it is extensible and not extended before (lines 8 to 14). Here, the method *Extensible* alleviates unnecessary interleaving among different sensors. After this, the leaf prefixes in the stack *workingLeaf* are popped for further extension, and the leaf prefixes of the resultant prefix are pushed to *workingLeaf*. This will be repeated until *workingLeaf* becomes empty. We can see from Algorithm 3 that a prefix is further extended (lines 19 to 26) only if it has not executed a global-unsafe task or a φ-unsafe action. Finally, p_i is assigned to the i^{th} element of the sensor cartesian vector *scv* (*scv*$[i]$) by line 29.

6.3.3 Sensor Prefix Generation

Algorithm 4 shows how a sensor $\mathcal{S}$ establishes a prefix from $\mathcal{C}$ w.r.t. φ. The following definitions are used in this algorithm.

- Method *getCurrentTsk*$(\mathcal{C}, \mathcal{S})$: return the currently executing task of $\mathcal{S}$ at state $\mathcal{C}$.
- Method *ExecuteTask*$(t, p, \varphi, \mathcal{C}, \mathcal{S})$: extending p by executing t w.r.t. φ. Details could

Algorithm 4 Prefix Generation

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

```
1:  p ← ⟨C⟩
2:  t ← getCurrentTsk(C, S)
3:  {extend p by executing task t}
4:  ExecuteTask(t, p, φ, {C}, S)
5:  if t terminates then
6:     for all p_i ∈ leaf(p) do
7:        C' ← last(p_i)
8:        irs ← GetItrs(C', S)
9:        p'_i ← RunItrs(C', itrs, S)
10:       p_i ← (p_i, {p'_i})
11:    end for
12: end if
13: return p
```

be found in Algorithm 5.

- Method *GetItrs*$(\mathcal{C}, \mathcal{S})$: return the set of pending interrupt requests of $\mathcal{S}$ at state $\mathcal{C}$.
- Method *RunItrs*$(\mathcal{C}, itrs, \mathcal{S})$: interleaving the execution of $\mathcal{S}$'s interrupt requests in *itrs* based on a persistent set approach.

The mission of Algorithm 4 is to generate a valid sensor prefix w.r.t. a certain property by executing the task enabled at a given state. Initially, p is initialized as $\langle\mathcal{C}\rangle$ (line 1) and the current enabled task is obtained by line 2. Then actions in t will be executed until t terminates or an φ-unsafe action is encountered, as shown in Algorithm 5. Interrupts are handled after a task has terminated, which is reasonable by Lemma 3 (lines 5 to 12). As for actions that post a task, interrupts are considered when executing such actions using a persistent set algorithm. More details will be discussed in Algorithm 6.

In Algorithm 5, 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 interrupts 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 3.

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*

Algorithm 5 Task Execution

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

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

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 from being stuck by loops.

6.3.4 Persistent Set Algorithm

In this section, we present the algorithms to apply partial order reduction at sensor level in order to minimize interleaving caused by interrupts. The idea is motivated by the observation that the only shared resource among interrupt handlers and normal actions is the task queue. One reason is that in the NesC language, variables are defined within a component's scope. The other reason is that only interrupt handlers are abstracted to exclusively access corresponding device status variables. By Lemma 3, we only need to consider two kinds of concurrency, i.e. the concurrency between a *post* statement, and the concurrency between any two interrupt handlers. This is implemented by Algorithms 6 and 7.

Algorithm 6 (*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 7 (Persistent Set) establishes a persistent set from a given set of actions *itrs*. If *itrs* contains a *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 actions in the persistent set.

Algorithm 6 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 of *p*}
18: $p \leftarrow (tr(p), br(p) \cup \{lp\})$
19: **end for**
20: **return** *p*

Algorithm 7 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*

6.4 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 9 and 10 assure the correctness of the functions invoked in Algorithm 4.

Lemma 9. *Given* $t \in Tasks(\mathcal{S})$, $t \in EnableT(\mathcal{C})$ *and* φ, $ExecuteTask(t, \langle \mathcal{C} \rangle, \varphi, \{\mathcal{C}\}, \mathcal{S})$ *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.* □

Lemma 10. *Given a network state* $\mathcal{C}$ *where* $\mathcal{C}[i] = (V, Q, B, \checkmark \triangle H)$, $RunItrs(\mathcal{C}, GetItrs(\mathcal{C}', \mathcal{S}_i))$ *terminates and returns a valid prefix of* $\mathcal{S}_i$. □

Based on Lemma 4 and 5, we can show the correctness of Algorithm 3 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 4 terminates and returns a valid prefix of $\mathcal{S}$ for some SNCV.*

Proof By Lemma 9, 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 $\mathit{leaf}(\widehat{p}) = (\mathit{leaf}(p) - p_i) \cup \mathit{leaf}(p_i')$ since p_i has been concatenated with p_i' and is no longer a leaf prefix. By Lemma 10, 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 9 and 10, both lines 3 and 8 terminates. Further, we assume that variables are finite-domain, and therefore the size of $\mathit{leaf}(p)$ is finite assuring that the for loop terminates. □

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

Proof We prove that at the beginning of each iteration of the while loop (lines 15 to 27) in Algorithm 3 the following conditions hold for any i $(1 \leq i \leq n)$:

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

By line 7, it is immediately true that $\mathit{first}(p_i) = \mathcal{C}$, and by Lemma 7, we can conclude that $\mathit{first}(p_i) = \mathcal{C}$ holds for all iterations. Since p_i is extended by executing $\mathit{GetPrefix}(\mathcal{S}_i, \widehat{\mathit{last}}(p_k), \varphi)$ (line 19), 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 8 to 14, we can immediately obtain that $\mathit{workingLeaf} = \{lp \in \mathit{leaf}(p_i) \mid \mathit{Extensible}(lp, \mathcal{S}_i, \varphi) \ \wedge \ \widehat{\mathit{last}}(lp) \notin \mathit{visited}\}$ 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 16, we can obtain that $\mathit{workingLeaf} = w_m - \{p_k\}$. By lines 19 to 26, $\mathit{wokingPrefix} = (w_m - \{p_k\}) \cup \{lp \in \mathit{leaf}(p_k') \mid \mathit{Extensible}(lp, \mathcal{S}_i, \varphi) \wedge \widehat{\mathit{last}}(lp) \notin \mathit{visited}\}$ (1). Let $\widehat{p}_k$ be the new value of p_k. After executing line 20, we have $\widehat{p}_k = (p_k, p_k')$. By Lemma 7, now we can obtain $\mathit{leaf}(\widehat{p}_k) = \mathit{leaf}(p_k')$ (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 $w_m = \{p \in \mathit{leaf}(p_m) \mid \mathit{Extensible}(p, \mathcal{S}_i, \varphi) \ \wedge \ \widehat{\mathit{last}}(p) \notin \mathit{visited}\}$, 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 $\forall\, t \in \mathit{tasks}(p_i), \alpha \in \mathit{acts}(p_i).\ t \not\subset_{GI} \mathcal{S}_i \Rightarrow t \in \mathit{LastT}(p_i) \ \wedge \ (\alpha \notin \mathit{safe}(\varphi) \vee \alpha \in \sum^{sd}) \Rightarrow \alpha \in \mathit{lastAct}(p_i)$ holds when the while loop terminates. Thus the cartesian vector generated by Algorithm 3 is valid.

For termination, we assume that all variables are finite-domain 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 2. On the other hand, Algorithm 3 uses *visited* to store each state that has been used to generate a new prefix, and by lines 10

and 23 a state is used at most once to generate a new prefix. Thus termination is guaranteed. □

Let φ be a property and ψ be the set of propositions belonging to φ. We say that two transition systems $\mathcal{T}$ and $\mathcal{T}'$ are 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 some trace σ' in $\mathcal{T}'$ such that σ and σ' are stuttering equivalent w.r.t. ψ, i.e., $\sigma \equiv_{st_\psi} \sigma'$, and vice versa. 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 $\sigma' = \mathcal{C}'_0, \alpha'_0, \mathcal{C}'_1, \alpha'_1, \cdots, \mathcal{C}'_i, \alpha'_i, \cdots$, the definition of two stuttering equivalent traces is then presented. $\sigma \equiv_{st_\psi} \sigma'$ 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.

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 in the following definition.

Definition 22 (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 state $\mathcal{C} = (V, Q, B, P)$, let $trExs(\mathcal{C}, Q)$ be the set of all possible traces after executing all tasks in Q. In the following, we define task sequences that generates stuttering equivalent traces. Given two states $\mathcal{C} = (V, Q, B, P)$ and $\mathcal{C}' = (V, Q', B, P)$, the two task sequences Q and Q' are stuttering equivalent w.r.t. φ $(Q \simeq_{st_\varphi} Q')$ iff $\forall \sigma \in trExs(\mathcal{C}, Q).\ \exists \sigma' \in trExs(\mathcal{C}', Q').\ \sigma \equiv_{st_\varphi} \sigma'$ and vice versa. The following lemma illustrates that *GetPrefix* returns a prefix of traces that are stuttering equivalent to those generated by the original semantics.

Lemma 11. *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 12. *Given a property φ, and two configurations $C = (V, Q, B, P) \in S$, $C' = (V, Q', B, P) \in S$ for all traces resulting of executing the actions starting from $\alpha_i = currentAction(P)$ and $\sigma = C = C_0, \alpha_0, \cdots, \alpha_i, \cdots, \alpha_{n-1}, c_n$ where for all j $0 \leq j \leq n-1$ $\alpha_i \notin \sum^{iq} \Rightarrow \alpha_j \in t$, and α_{n-1} is the last action from t or holds some of the conditions*

2., 3. from Definition 21, and $C_n = (V_\sigma, Q^\frown Q_\sigma, B_\sigma, P_\sigma)$ *then GetPrefix*(C') *returns a valid prefix p such that there exist a trace* p_i *in p,* $\sigma' = C' = C'_0, \alpha'_0, \cdots, \alpha'_i, \cdots, \alpha'_{n-1}, C'_n$, $C'_n = (V_\sigma, Q'^\frown Q'_\sigma, B_\sigma, P_\sigma)$ *where* $\forall j, 0 \leq j \leq n, \alpha_j \in \sigma_i.\alpha_j \notin \sum^{iq} \Rightarrow \alpha_j \in t$ *and* $\sigma \equiv_{st_\varphi} \sigma'$ *and if all the actions from t have been executed then* $Q_\sigma \simeq_{st} Q'_\sigma$.

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 $\alpha_j \in \sum^{iq}$, 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 5, *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 5 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 $\alpha_{int} \in \sum^{iq}$ 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 $i, 0 \leq i \leq n$ $\alpha_{\gamma_i} \in \sum^{iq}$ 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 $\alpha'_{p_k} \in \sum^{iq}$ 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 15 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 $\sigma_t = C_{t_0}$,α_{t_0},$\cdots$,α_{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 $\sigma_\varphi = C = C_{\varphi_0}$, α_{φ_0} ,$\cdots$,α_{φ_l},$C_{\varphi_{l+1}} = (V, Q^\frown Q'_p, B, P_\sigma)$ with $Q'_p = tsk(\alpha_{\varphi_0})^\frown \cdots ^\frown tsk(\alpha_{\varphi_n})$ and by (1) $Q'_p \simeq_{st} Q_p$. By Algorithms 5, 6 and 7 *GetPrefix*(C') returns a prefix p that contains a trace $\sigma' = C'$,α'_0,$\cdots$,α'_j,$C_{\alpha'_{j+1}}$, σ_φ,$\cdots$,α'_n,$C'_{n+1} = (V, Q^\frown Q'_\sigma, B, P_p)$ with $C_{\sigma_\varphi} = (V, Q^\frown Q_{\sigma_\varphi}, B, P_\gamma)$ 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 11 $Q''_p \simeq_{st} Q'_p$ 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 $C_\gamma = (V, Q^\frown Q_\gamma, B, P_\gamma)$ and $Q_\gamma = tsk(\alpha_{\gamma_0})^\frown \cdots ^\frown tsk(\alpha_{\gamma_n})^\frown t$. 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 $\sigma_\varphi = C = C_{\varphi_0},\ \alpha_{\varphi_0},\cdots,\alpha_{\varphi_l},\mathcal{C}_{\varphi_l},\alpha_\gamma,\ C'_\gamma = (V, Q^\frown Q'_\gamma, B, P_\varphi)$ 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 $Q'_\gamma = tsk(\alpha_{\varphi_0})^\frown \cdots ^\frown \alpha_{\varphi_{l-1}})^\frown t$. Therefore $Q'_\gamma \simeq_{st} Q_\gamma$. By Algorithms 5, 6 and 7 $GetPrefix(C')$ 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 $i,\ 0 \leq i \leq j\ \alpha'_i$ 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 $C_{\sigma_\varphi} = (V, Q^\frown Q''_\gamma, B, P_\gamma)$. Therefore $Q''_\gamma \simeq_{st} Q'_\gamma$ and $Q''_\gamma \simeq_{st} Q_\gamma$. By the I.H. $\sigma_\gamma = \mathcal{C}_\gamma,\alpha_{\gamma+1},\cdots,\alpha_{p_{m-1}},\ C_{p_m} = (V_p, Q^\frown Q'^\frown_\gamma Q_{\gamma_0}, B_p, P_p)$ there exist a sequence of tasks $\sigma'_\gamma = C_{\sigma_\varphi},\alpha_{p_0},\cdots,\alpha'_{p_{m-1}},\ C'_{p_m} = (V_p, Q^\frown Q''^\frown_\gamma Q'_{\gamma_0}, B_p, P_p)$ where $\cap Q'^\frown_\gamma Q_{\gamma_0} = Q_\sigma$, $Q''^\frown_\gamma Q'_{\gamma_0} = Q'_\sigma$ such that $Q'_{\gamma_0} \simeq_{st} Q_{\gamma_0}$. By By lemma 11 $Q_\sigma = Q'^\frown_\gamma Q_{\gamma_0} \simeq_{st} Q'_\sigma = Q''^\frown_\gamma Q'_{\gamma_0}$ and $Q_\sigma \simeq_{st} Q'_\sigma$. □

Lemma 12 shows that for all the possible local interleaving from $c\ \pi$ in the original network $\mathcal{N}$ starting and ending in t, one traversal t from the prefix returned by *GetPrefix* contains the same sequences of valuations for the set of propositions from φ.

Lemma 13. *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 21 then Algorithm 3 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 12 $GetPrefix(C'_0)$ 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 $t_j \in Tasks(S), t_i \in Tasks(S)$. $\alpha_i \notin \sum^{iq} \Rightarrow \alpha_i \in t_i\ \alpha_j \notin \sum^{iq} \Rightarrow \alpha_j \in t_j$ and $t_j \neq t_i$, or $\alpha_i \in \sum^{iq}, \alpha_j \in \sum^{iq}$ and $\alpha_i = \alpha_j$. Let $\sigma_i = C_0,\alpha_0,\cdots,\alpha_{i-1},C_i = (V_i, Q^\frown Q_\sigma, B_i, P_i)$. By Lemma 12 $GetPrefix(C'_0)$ returns a prefix p such that contains a trace $\sigma'_i = c' = c'_0,\alpha'_0,\cdots,\alpha'_i,\cdots,\alpha'_{n-1},c'_n,\ c'_n = (V_i, Q'^\frown Q'_\sigma, B_i, P_i)$ 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 $\mathcal{N}$, where $\mathcal{N} = (\mathcal{R}, \{S_0, \cdots, S_N\})$. Let $\mathcal{T}'$ be the transition system obtained after applying the two-level partial order reduction w.r.t. φ over $\mathcal{N}$. Then $\mathcal{T}'$ and $\mathcal{T}$ are stuttering equivalent w.r.t. φ.*

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 3, Theorem 3 and Lemma 12, 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 [102] 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 3, Theorem 3, and Lemma 12 $\sigma_i = \mathcal{C}_0, \cdots, \alpha_{k_1}, \cdots, \alpha_{k_{x+1}}, \mathcal{C}_{k_{x+1}+1} \equiv_{st_\varphi} \sigma'_i =$ $\mathcal{C}_0, \cdots, \alpha'_{k_1}, \cdots, \alpha'_{k_{x+1}}, \mathcal{C}'_{k_{x+1}+1}$ 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. φ, and if φ is an LTL-X property, then $\mathcal{T}' \models \varphi$ if and only if $\mathcal{T} \models \varphi$ [34]. This indicates that our method preserves LTL-X properties.

6.5 Experiments and Evaluation

In this section, we first study the performance of the two-level POR method by the Blink application to show how the state space is reduced. Then we evaluate the performance of the two-level POR method using a number of real-world SN applications. After that, a comparison between our POR and the POR implemented in T-Check [89] is provided, since T-Check provides verification of TinyOS/NesC programs with a POR algorithm. Experiments were conducted on a PC with Intel Core 2 Duo CPU (2.33GHz) and 3.25GB memory with Windows XP.

6.5.1 Example: the Blink Application

In this section, we demonstrates the reduction achieved by our two-level POR algorithm by the Blink application presented in Example 6 presented in Chapter 4.

The original state space of Blink is shown in Figure 6.2(a), where the timing interrupt event $Sensor1.Timer0.\mathit{firing}$ has introduced a lot branches. In Blink application, toggling the red LED is always independent with timer interrupt since there is no common accessed resource. Therefore, the interleaving between firing event and other event can be pruned, resulting in the state graph in Figure 6.2(b), which is automatically generated by our tool. It is obvious that originally there are 22 states and after sensor-level partial order reduction there are only 16 states.

Suppose that there is a network with two sensors both of which are running the Blink application. Since there is no network communication between sensors in Blink application, it is easy to calculate that the total number of states in the network is $22^2 = 464$. If only sensor-level reduction is applied, then the total number of states becomes $16^2 = 256$. However, we could still decrease this number by applying reduction to the network level. Since there is no communication between the two sensors, every task inside a sensor becomes *global independent*. Therefore, the interleaving between sensors could be delayed until a sensor prefix could not be further extended, i.e., when a maximal prefix is obtained. After network-level reduction, the state space is reduced to be only 64 states, as shown in Figure 6.3.

Given a network $\mathcal{N}$ with n sensors $\mathcal{S}_1, \mathcal{S}_2, \cdots, \mathcal{S}_n$, let S_i be the size of sensor S_i. Then the size of the network will be $N \approx \times_{i=1}^{n} S_i$.

Let $\widehat{S}_i$ be the size of sensor $\mathcal{S}_i$ after sensor level reduction, and let k_i be the global dependent factor of $\mathcal{S}_i$, representing the number of sensor prefixes that $\mathcal{S}_i$ has, which is actually the number of necessary interleaving of sensor i. Then the total number of prefixes is $\sum_{i=1}^{n} k_i$). Since our algorithm establishes the system state space by interleaving prefixes instead of states, now we calculate the total number of branches after fully exploring the state space. Intuitively, the "full" execution will have the branches of the total order of all prefixes, which has $(\sum_{i=1}^{n} k_i)!$ orderings. However, this is an over-approximation, because

(a) w/o POR (b) w/t POR

Figure 6.2: State Graph of a Blink Sensor

that for those prefixes of the same sensor there is only one order. We can know that the number of unnecessary orderings of each sensor's prefixes is $(k_i!)$. Thus, the actual total number of branches should be $\frac{(\sum_{i=1}^{n} k_i)!}{\times_{i=1}^{n}(k_i!)}$, which, however, is still an approximation, since we consider the prefix of each sensor as a sequential trace during the calculation. Since each branch is in fact a total execution path of the network, then the number of states in each branch is $\sum_{i=1}^{n} \widehat{S}_i$. Therefore, the size of the network after the two-level POR approach will be $\widehat{N} \approx (\sum_{i=1}^{n} \widehat{S}_i) \times \frac{(\sum_{i=1}^{n} k_i)!}{\times_{i=1}^{n}(k_i!)}$ ($n!$ is the factorial of n). Then the reduction ratio $\widehat{R}$ of two-level POR is $\widehat{R} \approx (1 - \frac{\widehat{N}}{N})$, based on which we can obtain that $\widehat{R} \propto (\varrho_i, \frac{1}{k_i})$, where ϱ_i is the local dependent factor of $\mathcal{S}_i$, i.e., $\varrho_i = 1 - \frac{\widehat{S}_i}{S_i}$. Ideally, k_i should be 1, which means that there is no global dependent action between sensors and the reduced network size becomes $\widehat{N} = n \times \sum_{i=1}^{n} \widehat{S}_i$. And the Blink example is an ideal case.

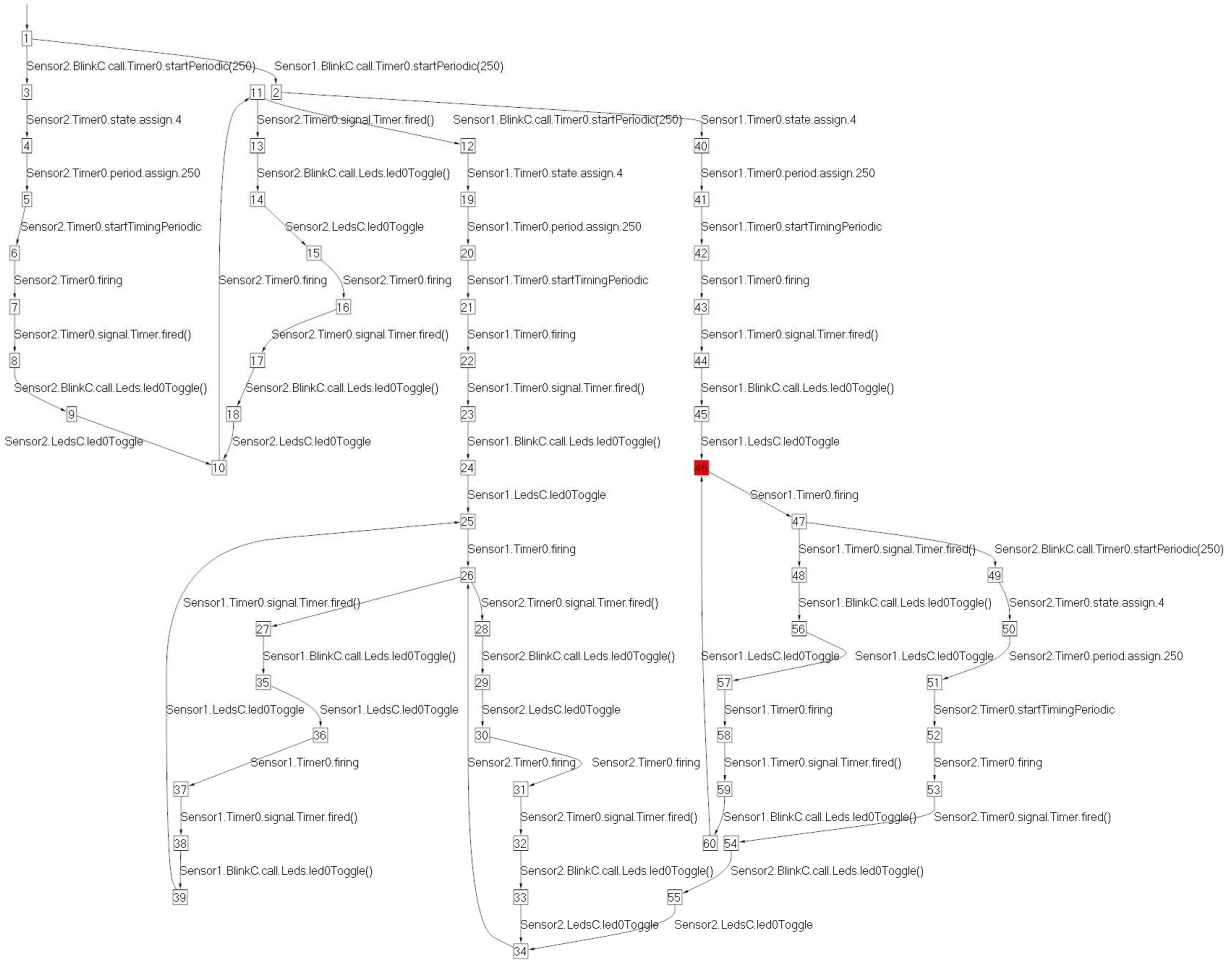


Figure 6.3: State graph of a 2-node Blink network after POR

6.5.2 Enhancing NesC@PAT with Two-level POR

In this section, we present the experiment results obtained by the two-level POR. In particular, we study two real-world applications, an anti-theft application and the Trickle algorithm [88]. 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. Besides, we also considered the following TinyOS applications as benchmarks:

- Blink: an extended version of the simple Blink application in Section 6.5.1, where three timers are used to periodically toggle three LEDs, respectively.
- BlinkTask: a modified version of the simple Blink application where a task is posted to toggle the red LED instead of calling the LED toggle command directly in the timer

App(LoC)	**Property**	**Size**	**Result**	**#State**	**#Trans**	**Time(s)**	**OH(ms)**	**#States** *wo POR*	**POR Ratio**
Blink(30)	Non-termination	1	✓	50	50	0.1	0.05	360	0.13
		2	✓	475	500	0.1	0.06	129K	3×10^{-3}
		3	✓	3500	3750	0.36	0.08	46.6M	3×10^{-5}
		4	✓	23K	25K	2	0.09	16796M	1×10^{-6}
		6	✓	859K	937K	79	0.14	2.17e+15	4×10^{-10}
BlinkTask (20)	Non-termination	1	✓	20	20	0.02	0.03	32	0.6
		2	✓	144	160	0.01	0.04	1024	0.14
		3	✓	832	960	0.06	o.13	32K	0.02
		4	✓	4352	5120	0.30	0.07	1M	4×10^{-3}
		6	✓	908K	988K	292	12	1073M	8×10^{-4}
Example (233)	Non-termination	2	✓	2040	2129	0.1	1.18	45K	0.04
		3	✓	30K	31827	1.4	1.30	9M	3×10^{-3}
		4	✓	276K	294K	14		2025M	1×10^{-4}
		6	✓	2.3M	2.5M	129	30	9.11e+13	2.5×10^{-8}
Anti-theft(3391)	Non-termination	3	✓	57.4K	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	✓	1848	1880	0.4	2	111683	1×10^{-2}
		3	✓	48K	50K	9	3	>23.7M	$< 2 \times 10^{-3}$
		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 6.1: Performance of Two-level POR

fired event.

- Example: the running example shown in Figure 5.10(a).

For the anti-theft application, we checked if a sensor turns on its theft LED whenever a theft is detected, i.e., □(theft ⇒ ◇alert), based on the following propositions.

- theft ≡ NodeA.AntiTheftC.detected==1;
- alert ≡ NodeA.AntiTheftC.ledon==1.

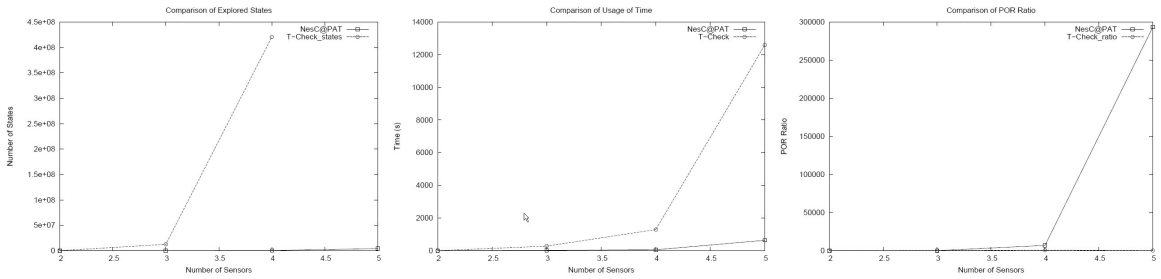
Then the property □(theft ⇒ ◇alert) is specified as (#assert SensorNetwork |= [](theft -> <> alert)) in NesC@PAT.

In the Trickle algorithm, we checked that eventually all the nodes are updated with the latest data among the network, i.e., ◇*AllUpdated*. We also checked a safety property to guarantee that each node never performs a wrong update operation. The properties of Trickle algorithm are specified in the same way as shown in Section 5.6.

Both applications as well as the benchmarks are verified using NesC@PAT against the aforementioned properties with the two-level POR, on a PC with Intel Core 2 Duo CPU (2.33GHz) and 3.25GB memory with Windows XP. Results are presented in Table 6.1. The results of the safety property against the Trickle algorithm are presented in Table 6.2 for the comparison with T-Check.

Size	NesC@PAT					T-Check					
	wt POR			#State *wo POR*	Ratio	#B	*wt POR*			#State *wo POR*	Ratio
	#S	Exh	T(s)				#S	Exh	T(s)		
2	1683	Y	2	52.3K	3×10^{-2}	20	4765	Y	1	106.2K	$\approx 4 \times 10^{-2}$
3	27K	Y	20	>11.8M	$< 2 \times 10^{-3}$	12	66.2K	N	1	13.5M	$\approx 5 \times 10^{-3}$
						50	12.6M	Y	283	NA	NA
4	192K	Y	58	>2.7G	$< 7 \times 10^{-5}$	10	56.7K	N	1	41.8M	$\approx 1 \times 10^{-3}$
						50	420.7M	Y	1291	NA	NA
5	2.1M	Y	638	>616G	$< 3 \times 10^{-6}$	8	85.2K	N	1	17.4M	$\approx 1 \times 10^{-3}$
						50	NA	N	>12600	NA	NA

Table 6.2: Comparison with T-Check



(a) States Explored (b) Time Used (c) Reduction Ratio

Figure 6.4: Comparing NesC@PAT with T-Check

In Table 6.1, 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 6.1 and 6.2) is also an under approximation. With this under-estimation, our POR approach achieves a reduction of at least 10^2-10^{10}. Further, the larger a network is, the more reduction it will be.

6.5.3 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., $POR\ Ratio \approx \frac{\#State\ wt\ POR}{\#State\ wo\ POR}$, 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 6.2, where *Exh* indicates if all states are explored and #**B** is the bounded number used for stateless model checking by T-Check. 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.

6.6 Summary

In this chapter, we have presented a novel two-level POR for sensor networks, which has been integrated into NesC@PAT. We have studied how the two-level POR improved NesC@PAT's performance by a number of examples, and results have indicated that our POR approach outperforms an existing POR approach for SNs. In the following chapter, we discuss the tool of our work, NesC@PAT, which incorporates the approaches presented in this thesis, including the NesC2STCSP approach, the direct verification approach and the two-level POR.

Chapter 7

NesC@PAT

Our approach has been implemented and integrated as the NesC module in the model checking framework PAT [97, 144, 99], named NesC@PAT[1]. NesC@PAT is a fully automatic model checker, which takes NesC programs and a network topology as the input and verifies the SN against both safety and liveness properties specified as state reachability or linear temporal logic (LTL) formulas. The expressive power of LTL has allowed the tool to specify a larger set of properties, compared to that supported by other similar tools like T-Check [89].

NesC@PAT covers most of the C-like and NesC features including pointers, commands, events and tasks or even advanced features like parameterized wiring and hierarchical wiring. Currently it is capable of analyzing a large set of NesC programs that execute on real motes, without any modification to the code. The development of NesC@PAT started in Nov, 2009, and its core implementation exclusive of PAT and GUI contains 153K lines of C# code, as shown in Table 7.1.

Component	# **LoC**	**Summary**
Parser	100K	Generated from 9K LoC of grammar rules by a grammar tool
Model Generator	50K	Encoding the formal semantics of SNs and the two-level POR algorithm
POR Engine		
NesC2STCSP	1K	Integrating the transformation rules from NesC to STCSP
PAT Related	2K	Linking PAT's GUI, editor, simulator and verification algorithms
Total	153K	

Table 7.1: Distribution of LoC

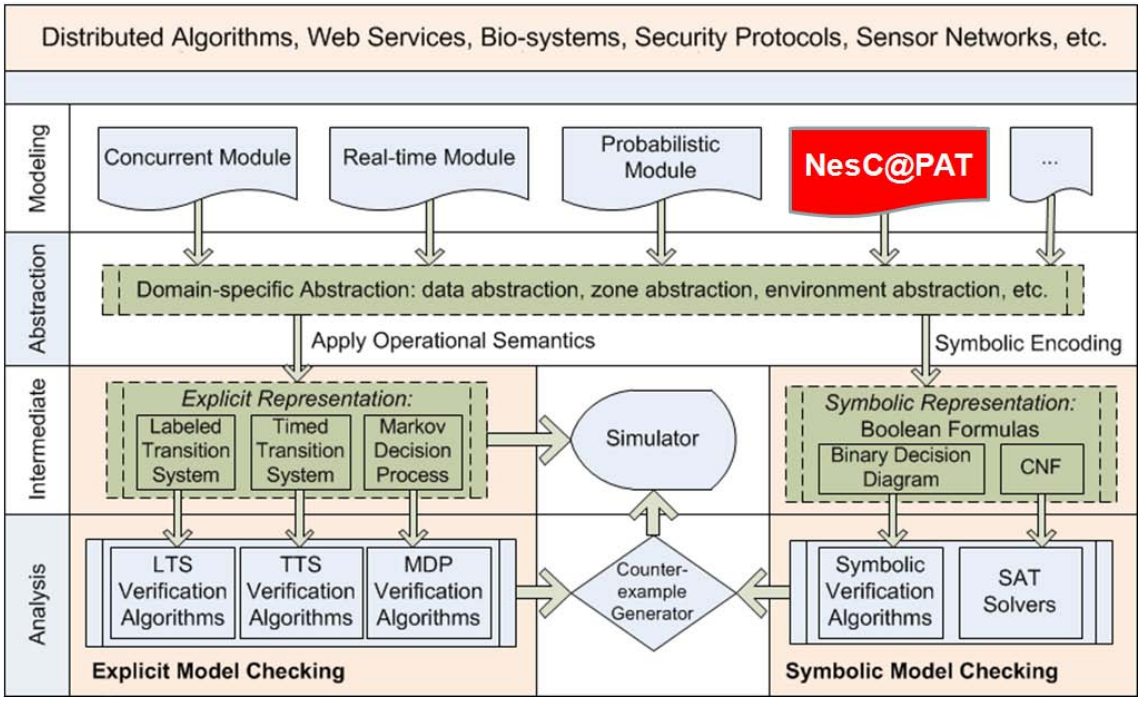


Figure 7.1: PAT Framework

7.1 PAT Model Checking Framework

PAT (Process Analysis Toolkit) [97, 146, 99, 100] is a generic and extensible framework that facilitates effective incorporation of domain knowledge with formal verification using model checking techniques. PAT implements a library of model checking techniques catering for checking deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions [147, 144], refinement checking [141, 95, 160, 94] and probabilistic model checking. PAT has been implemented with advanced optimization algorithms like process abstraction, partial order reduction, symmetry reduction and so on [98, 149, 169]. Besides explicit model checking, PAT supports symbolic model checking as well, providing a BDD library to support different domains [117, 119, 118]. PAT has been used for studying various algorithms and systems [168, 96, 167, 155, 135, 140, 138, 93, 71, 136].

As shown in Figure 7.1, PAT is a self-contained environment to support composing, simulating and reasoning about system models. It comes with user friendly interfaces, a featured model editor and an animated simulator. PAT has been extended for modeling and verifying various domain-specific systems like Communicating Sequential Process (CSP#) [142, 44], Stateful Timed CSP [148, 99, 143], C# language [174], Orchestration language [152, 153], probabilistic CSP (PCSP) [137, 151], UML [166], security protocols [154, 103], web ser-

[1]available at http://www.comp.nus.edu.sg/~pat/NesC/

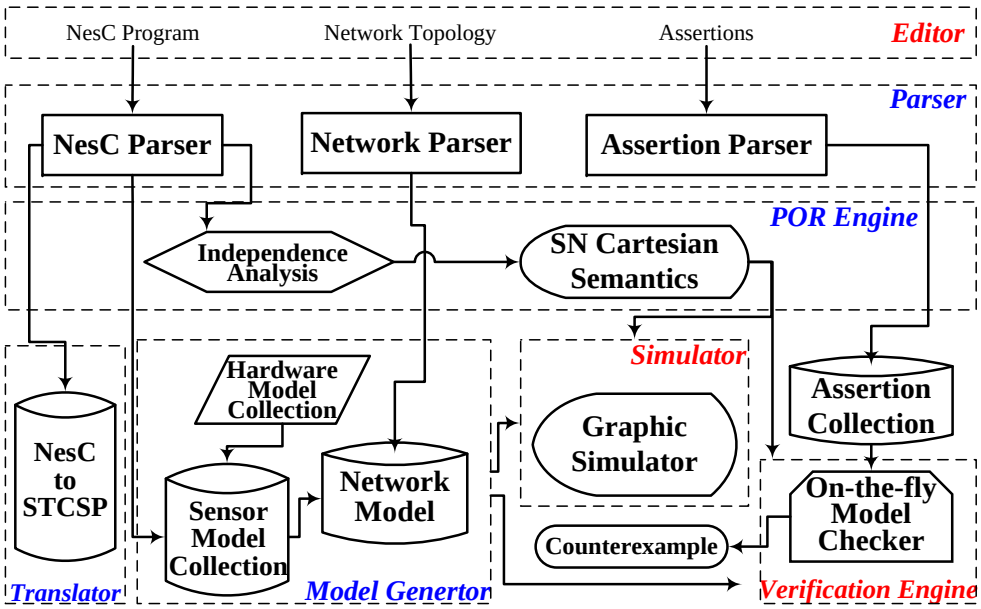


Figure 7.2: Architecture of NesC@PAT

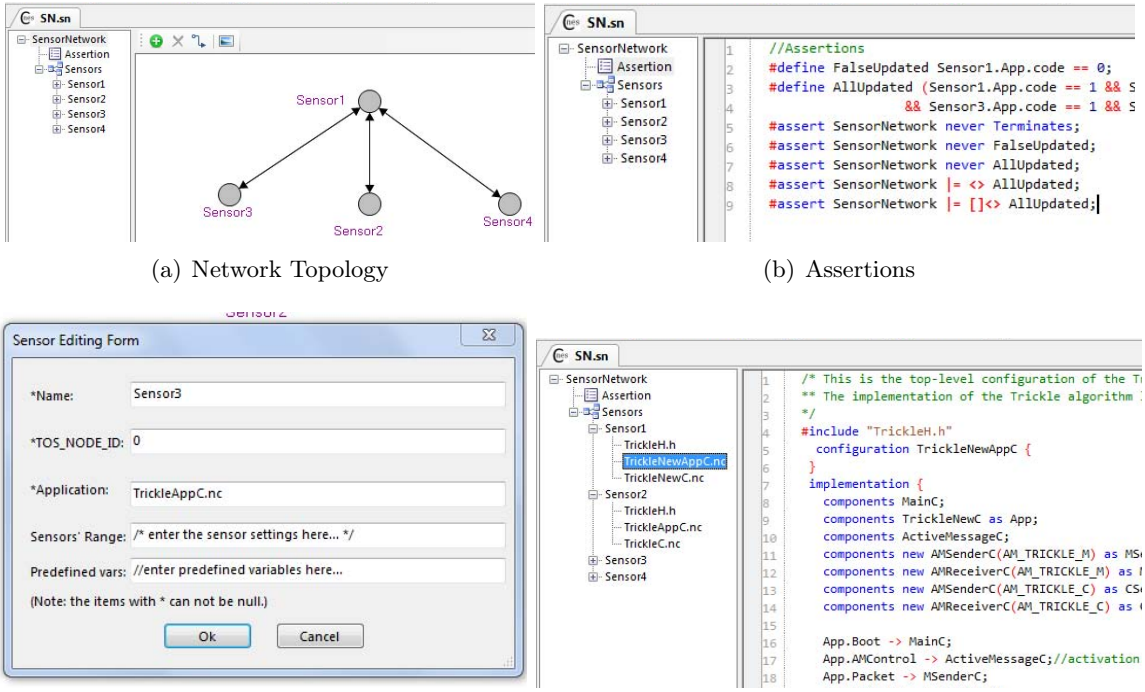
vice [145], PRTS (probability real-time system) [150, 139], ERA (Event-Recording Automata) [91], PDDL [90], timed system [92], stateflow diagrams [29], etc. NesC@PAT is implemented as the NesC module in PAT, which will be explained in next section.

7.2 System Introduction

Figure 7.2 illustrates the architecture of NesC@PAT. There are seven components, *i.e.* the Editor, the Parser, the POR Engine, the NesC2STCSP Translator, the Model Generator, the Simulator, and the Verification Engine. For the implementation of the Editor, the Simulator and the Verification Engine, NesC@PAT reuses the existing implementation of the PAT framework and the implementation effort for these components has been reduced to only linking the computation of NesC@PAT with the corresponding library provided by PAT. In the following, we illustrate each of the seven components in detail.

7.2.1 Editor

The **Editor** provides a Visual-Studio-like editing environment to input NesC programs running on sensors, as shown in Figure 7.3(d). Different sensors are allowed to run different copies of NesC programs in an SN. The network topology is presented in the form of a directed graph, and the Editor offers a graphical GUI for drawing topology graphs, as illustrated in Figure 7.3(a). The communication link between sensors are presented in the form of directed arrows.



(a) Network Topology (b) Assertions

(c) Sensor Configuration (d) NesC Programs

Figure 7.3: The Editor of NesC@PAT

The Editor also allows users to switch to assertion view or program view to edit assertions (Figure 7.3(b)) or NesC programs (Figure 7.3(d)), respectively. In this way, verification goals can be defined without modifying the NesC program. This is desirable because it makes it possible to directly deploy the verified NesC programs in practice, without any further changes introduced by verification.

As shown in Figure 7.3(c), a sensor can be configured with the following parameters: name, *TOS_NODE_ID* (i.e., the unique ID of the sensor referred to by TinyOS), location of the NesC program (i.e., the NesC file that implements the top-level configuration), the data ranges of sensing components and predefined variables. It is important to specify a data range for each sensing component (if any) of a sensor, otherwise the system space will become extremely large or even infinite, making it infeasible for verification tasks.

Moreover, NesC@PAT also provides an interface for advanced configuration, which allows users to define the hardware platform, task queue size, buffer size, etc. The hardware platform is chosen to be IRIS by default. As for the task queue size, it is set to be 256 by default, which is the default size of the task queue in TinyOS. The message buffer is set as 1 by default. This is to reduce the states introduced by networked communication. In NesC@PAT, a message packet will be abandoned when the message buffer is full and in this way the packet loss feature of SNs is captured. However, this buffer size needs to be larger sometimes in order to avoid false-negatives.

7.2.2 Parser

The **Parser** compiles all inputs from the editor, including the network topology, the NesC programs of each sensor and the assertions to be verified. NesC@PAT has supported the parsing of interface files, module files, configuration files of NesC programs as well as header files with the file extension of ".h" files. A sensor network model is stored as a file with ".sn" extension including the network topology, the application path of each sensor node and the assertions to be verified.

7.2.3 Model Generator

The **Model Generator** is provided to generate models from NesC code automatically, for the aim of minimum manual effort. Generated models preserve the interrupt-driven concurrency among tasks and events, which is essential for TinyOS applications. In this way, the NesC program running on a node, like the fragment shown in Figure 1.1(a), is translated into a Label Transition System (LTS), avoiding manual construction of models and making NesC@PAT useful in practice.

The operational semantics for each NesC language structure provides the basis for constructing LTSs from NesC programs, which has been presented in Chapter 5. Moreover, the interrupt-driven feature of the TinyOS execution model is preserved in the generated LTS, which allows concurrency errors among tasks and interrupts to be detected. Radio, timer, sensor device and other devices are abstracted in the Hardware Model Collection, which is also taken into account in the generation of LTS. With the network topology specified in the Editor and individual node LTSs, the LTS of an SN is composed, considering the non-determinism between sensor nodes.

7.2.4 Verification Engine

The **Verification Engine** conducts an exhaustive search (optimized by partial order reduction) of the generated LTS state space, and it returns a counterexample if an assertion (i.e., a correctness criterion) is violated. Currently, it integrates model checking algorithms for verifying termination, state reachability and LTL formulas [144]. This allows flexibility for specifying significant goals to verify SNs against. Taking the code in Figure 1.1(a) as an example, the statement *post sendTask*() might not be able to execute, if a previous *sendTask* has not been posted successfully and thus *sendTaskBusy* remains *TRUE*. Such a scenario is undesirable and should be avoided under any circumstance. With NesC@PAT, it is very convenient to find this buggy behavior by model checking the code with the LTL property $\Box\Diamond sendTaskBusy = FALSE$ (**P1**), i.e. *sendTaskBusy* is *always eventually* set to *FALSE*.

NesC@PAT supports both sensor-level and network-level verification, against properties defined as state reachability or LTL properties. Non-termination and state reachability are checked by exhaustively exploring the state space using Depth-first search or Breadth-first

search algorithms. We adopt the approach presented in [144] to verify LTL properties. First, the negation of an LTL formula is converting into an equivalent Büchi automaton; and then accepting strongly connected components (SCC) in the synchronous product of the automaton and the model are examined in order to find a counterexample. Notice that the SCC-based algorithm allows us to model check with fairness [149], which often plays an important role in proving liveness properties of SNs.

7.2.5 Simulator

Using the **Simulator**, users can easily simulate the visualized execution of a node or a whole network step by step. At each step only a fine-grained statement (e.g., updating a variable) is executed, which provides detailed runtime behavior to be monitored. Moreover, if a property is violated, the Model Checker will report a counterexample to the simulator, so that users can reason about it and correct the buggy code. Verifying the code in Figure 1.1(a) against **P1**, the Model Checker will return a counterexample, in which there is a state where *post sendTask*() fails. Simulating this counterexample helps to correct the program by the revised code in Figure 1.1(b), where *sendTaskBusy* is set to *FALSE* if the statement *post sendTask*() fails.

7.2.6 POR Engine

Algorithm 8 DFS with POR

ParseSN($\mathcal{N}$)

```
1: {parse and statically analyze each sensor}
2: for all sensors S ∈ N do
3:     Parse(S)
4:     StaticAnalysis(S, φ)
5: end for
6: {depth-first search for state reachability}
7: vs ← {C0}
8: ws ← {C0}
9: setPfx(C0, ⟨C0⟩)
10: while ws ≠ ∅ do
11:     C ← ws.Pop()
12:     if C violates φ then
13:         return false
14:     else
15:         ns ← GetSuccessors(C, φ)
16:         for all C' ∈ ns do
17:             if C' ∉ visited then
18:                 ws ← ws.Push(C')
19:                 vs ← vs ∪ {C'}
20:             end if
21:         end for
22:     end if
23: end while
24: return true
```

The two-level POR approach presented in Chapter 6 has been implemented as the **POR Engine** in NesC@PAT. First, independence analysis of actions at sensor level and network level is performed statically at compile time. Second, the sensor network cartesian semantics is applied to generate the reduced state space before the model checking. Our POR approach is static, i.e., it can be directly integrated to existing model checking algorithms, without

Algorithm 9 DFS with POR

$MCwtPOR(\mathcal{N}, \mathcal{C}_0, \varphi)$

```
1: {parse and statically analyze each sensor}
2: for all sensors S ∈ N do
3:     Parse(S)
4:     StaticAnalysis(S, φ)
5: end for
6: {depth-first search for state reachability}
7: vs ← {C_0}
8: ws ← {C_0}
9: setPfx(C_0, ⟨C_0⟩)
10: while ws ≠ ∅ do
11:     C ← ws.Pop()
12:     if C violates φ then
13:         return false
14:     else
15:         ns ← GetSuccessors(C, φ)
16:         for all C' ∈ ns do
17:             if C' ∉ visited then
18:                 ws ← ws.Push(C')
19:                 vs ← vs ∪ {C'}
20:             end if
21:         end for
22:     end if
23: end while
24: return true
```

making significant modifications. As an example, we present the DFS algorithm with POR for checking safety properties as shown in Algorithm 9, where the following notations are used:

- Method $Parse(\mathcal{S})$: parsing a sensor $\mathcal{S}$.
- Method $StaticAnalysis(\mathcal{S}, \varphi)$: analyzing the independence relation of actions and tasks of sensor $\mathcal{S}$ w.r.t the property φ.
- Variable vs: the set of visited states.
- Variable ws: the stack of states that are to be further explored.
- Method $setPfx(\mathcal{C}, p)$: assigns prefix p as the prefix that state $\mathcal{C}$ belongs to.

During the compilation of the network model $\mathcal{N}$, independence analysis is applied to each sensor $\mathcal{S}$ (lines 2 to 5). The depth-first search (DFS) is implemented by lines 10 to 23, during which $GetSuccessors(\mathcal{C}, \varphi)$ is invoked for obtaining successors of each state and to establish a reduced state space. The DFS search will terminate immediately if a state violating the property is evidenced; otherwise, the DFS will terminate only and if only no more new states can be generated, in which case φ is considered as *valid* for the network $\mathcal{N}$.

Algorithm 10 statically identifies the dependent relation among tasks, in which the following notations are used:

- Variable $TD : Tasks \times Tasks$, the pairs of locally dependent tasks. Formally, $(t, t') \in TD \Leftrightarrow t \not\equiv_{TI} t$.
- Variable GD, the set of globally dependent tasks. Formally, $GD \subseteq Tasks \wedge t \in GD \Leftrightarrow t \not\subset_{GI} \mathcal{S}$.

Algorithm 10 Statical Analysis of Independence

StaticAnalysis$(\mathcal{S}, M, \varphi)$

1: $tks \leftarrow Tasks(\mathcal{S})$
2: $GD \leftarrow \varnothing$ {global dependent tasks}
3: {analyze global dependence of tasks}
4: **for all** task $t \in tks$ **do**
5: **if** $t \not\subset_{GI} \mathcal{S}$ **then**
6: $GD \leftarrow GD \cup \{t\}$
7: **end if**
8: **end for**
9: $TD \leftarrow \varnothing$ {task dependence relation}
10: {analyze local dependence among tasks}
11: **for all** task $t \in tks$ **do**
12: $tsk \leftarrow tsk - \{t\}$
13: **for all** task $u \in tsk$ **do**
14: **if** $t \not\equiv_{TI} t'$ **then**
15: $TD \leftarrow TD \cup \{(t, t'), (t', t)\}$
16: **end if**
17: **end for**
18: **end for**

7.2.7 Translator

The NesC2STCSP approach presented in Chapter 4 is implemented as the **Translator** component in NesC@PAT. The translator generates STCSP models from NesC programs for a certain sensor, and store the model in a file. Then one can use the PAT model checker to load the STCSP model from the file and perform verification tasks. The translator provides the complementary checking of timed features to the direct verification approach. Moreover, it provides a comparison between the performance of the NesC2STCSP approach and the direct verification approach.

Chapter 8

Conclusion

In this thesis, we developed approaches and techniques to verify sensor network (SN) implemented on TinyOS/NesC. The work consists a set of mapping rules from NesC language to STCSP, a complete formal semantics for networked NesC programs and SNs, and the two-level Cartesian partial order reduction (POR) for SNs. We have integrated our techniques into the systematic model checker for sensor networks, NesC@PAT. We used NesC@PAT to verify a number of real-world sensor network applications, protocols and algorithms, with or without POR settings. We have evaluated the performance of NesC@PAT and shown that NesC@PAT efficiently verifies SN implementations and detects errors or bugs effectively. We compared our tool with a similar tool T-Check and results show that our approach outperforms it. Our work can help SN developers to find bugs or errors in their systems before deployment and thus is significant for improving SN quality and reliability.

In Chapter 4 we presented the NesC2STCSP approach that obtains STCSP models from NesC programs automatically, which contains a set of mapping rules from NesC to STCSP. NesC2STCSP has been implemented in NesC@PAT, and has been used to study several NesC programs. We believe that this approach contributes to the robustness of sensor network systems because it allows NesC programmers to model check their code without being troubled by manually constructing formal models. However, due to the gap between the semantics of NesC and those of formalisms, formal models extracted from NesC are inevitably large, complex and redundant. Intuitively, direct verification is straightforward and more efficient as specialized optimization techniques are possible. Therefore, we came up with the direct verification approach explained in Chapter 5.

In Chapter 5 we presented the formal definitions of SNs, which were the basis for automatically generating the state space of SNs. Here, we highlight that the formal definitions of SNs have the following significance. First, the formal semantics defined in this thesis for NesC opens chances for the research community to directly verify SN implementations. Existing work [68, 89, 24] was either built based on simulators of TinyOS applications or translated NesC programs into some intermediate representation. According to our knowledge, our work is the first to define the complete semantics formally for networked NesC

programs. Second, our approach is the first to model the interrupt-driven execution model of TinyOS. This is important since it allows concurrency errors at sensor level to be detected. However, our work currently adopts a non-threaded execution model of TinyOS. Recently, new models have been proposed, e.g., TOSThread [79] has been proposed to allow user threads in TinyOS. Nevertheless, our work would still benefit the SN development since many SNs are implemented in non-threaded settings. Althouth the current component model library of NesC@PAT only models a subset of the TinyOS component library, our approach can be easily extended to support more component models. A future direction would be to design approaches for modeling TOSThread.

This direct verification approach has been integrated into NesC@PAT, making it the first self-contained model checker specifically for the NesC language, which systematically supports verification of both safety and liveness properties. We studied the behaviors of a Trickle algorithm by NesC@PAT and the results confirmed that NesC@PAT is able to find bugs thoroughly in the implementation of the algorithm. However, verifying a network with four nodes encountered an OutOfMemory exception in our testbed. This means that NesC@PAT is highly limited with the number of nodes and number of LOC of SNs. However, NesC@PAT opens up new avenues for the direct and automatic verification of SN implementations. In practice, there are usually a large number of sensors in a network and a real-world NesC program usually consists of thousands of LOC. Thus, there is a need to develop techniques to improve the scalability of NesC@PAT, which leads to the proposal of the two-level partial order reduction (POR) for SNs.

In Chapter 6, we presented the two-level POR algorithm, which was integrated into NesC@PAT based on independence analysis and the sensor network Cartesian semantics. We also proved that our POR is sound for verifying LTL-X properties. Experiments of verifying sensor networks with NesC programs of more than three thousands LOC were conducted with NesC@PAT under the POR setting. As shown in Section 6.5, we achieved reduction of more than 10^2 - 10^{10} for our test cases. This indicates that the two-level POR has significantly improved the efficiency of verification, by allowing sensor networks with thousands of LOC in each sensor to be model-checked exhaustively. To the best of our knowledge, the two-level POR proposed for SNs is the first to reduce the state pace of SNs at both global and local situations, and achieves a larger portion of reduction than existing approaches [89]. This approach could be generalized for reduction of systems of multiple levels of concurrency, and would be applicable for distributed systems, concurrent embedded systems besides SNs. It must be noted that the state space of SNs still grows quite fast with the number of sensor nodes. However, most errors and bugs in implementation can be found in SNs within ten nodes and thus verification results are satisfactory for most SNs with POR.

The contributions of our work are two-fold. First, the formal definitions of SNs and the two-level POR algorithm are theoretically significant for both researchers in the model

checking community to develop verification techniques for either SNs or other concurrent systems. Second, the model checker NesC@PAT can be applied directly and conveniently by SN developers to verify their programs before system deployment and thus will help improve the reliability of SNs.

Although we have achieved the above contributions, our work is still limited in the following aspects.

First, although the two-level POR approach has achieved substantial reduction of the state space for sensor networks, our approach is still incapable of handling large sensor networks of hundreds or thousands of nodes. Although some work has shown that a finite number of nodes are sufficient for analyzing certain properties of a network (e.g., [39]), a common problem is that the protocol properties might vary with the size of the network. For example, in [49], it was reported that errors found for the LMAC protocol were different between sensor networks with three, four and five nodes. One possible solution is to adopt techniques that reduce the verification of a large system to a smaller one like parameterized model checking techniques [32, 1]. Therefore, one of our future direction is to employ parameterized model checking techniques to support the verification of larger sensor networks.

Second, as shown by the Trickle algorithm in Section 5.6, the errors of a protocol might only exist in certain topologies. Therefore, it is interesting to verify sensor networks with different topologies. However, the number of possible topologies grows exponentially fast, i.e. $2^{N(N-1)}$ where N is the number of nodes. For example, a network with 5 nodes has 1024 topologies. This introduces an interesting but highly challenging to verify sensor networks with all possible topologies. Therefore, another future work direction is to work on verification techniques to solve this problem.

The third future direction is to develop local reasoning techniques for SNs. Cohen et. al. proposed to explore the behaviors and correctness of concurrent systems by only examining certain local behaviors of individual sub-systems or processes [113, 35, 36, 37]. Moreover, Guerraoui and Yabandeh discussed a local approach for model checking a networked system without the network [64]. Only the local nodes' states are explored in a separate manner and the set of valid system states is a subset of all combinations of the local states of nodes and checking validity of such a combination is only performed a posteriori, in case of a possible bug. This approach drastically reduces the number of transitions executed by the model checker. It is interesting to explore such techniques to SNs and to support the verification of large networks or even parameterized sensor networks.

Fully automatic model extraction from protocol implementation without relying on a pre-existing protocol specification is meaningful. Aizatulin et. al. verified cryptographic protocols coded in C for correspondence properties with respect to the computational model of cryptography [2, 3, 4, 5]. They first used symbolic execution to extract a process calculus model from a C implementation of the protocol. Then the extracted model

would be translated to a CryptoVerif protocol description, such that successful verification with CryptoVerif implies the security of the original C implementation. They successfully verify over 3000 LOC and the largest example is about 1000 LOC, during the analysis of which a vulnerability was uncovered. Recently, Bai et. al. proposed [9] AUTHSCAN to automatically extract specifications from protocol implementations and checking security flaws. AUTHSCAN recovers the authentication protocol specification from its implementation without any knowledge of the protocol specifications being checked. One more future direction of this thesis is to explore such techniques and to develop a tool for automatic analysis of SN security protocols at implementation level. We believe that this could be done with acceptable efforts since the formal semantics of SN source code is already defined in this thesis.

Bibliography

[1] Parosh Aziz Abdulla, Ahmed Bouajjani, Bengt Jonsson, and Marcus Nilsson. Handling Global Conditions in Parameterized System Verification. In *CAV*, pages 134–145, 1999. 8

[2] Mihhail Aizatulin, François Dupressoir, Andrew D. Gordon, and Jan Jürjens. Verifying Cryptographic Code in C: Some Experience and the Csec Challenge. In *Formal Aspects in Security and Trust*, pages 1–20, 2011. 8

[3] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution. In *ACM Conference on Computer and Communications Security*, pages 331–340, 2011. 8

[4] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Extracting and Verifying Cryptographic Models from C Protocol Code by Symbolic Execution. *CoRR*, abs/1107.1017, 2011. 8

[5] Mihhail Aizatulin, Andrew D. Gordon, and Jan Jürjens. Computational Verification of C Protocol Implementations by Symbolic Execution. In *ACM Conference on Computer and Communications Security*, pages 712–723, 2012. 8

[6] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. Wireless Sensor Networks: A Survey. *Computer Networks*, 38:393–422, 2002. 1

[7] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-Order Reduction in Symbolic State-Space Exploration. *Formal Methods in System Design*, 18(2):97–116, 2001. 3.4

[8] Will Archer, Philip Levis, and John Regehr. Interface contracts for TinyOS. In *IPSN*, pages 158–165, Massachusetts, USA, 2007. 1, 3.3

[9] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. AuthScan: Automatic Extraction of Web Authentication Protocols from Implementations. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, San Diego, CA, USA, February 2013. 8

[10] Christel Baier, Pedro R. D'Argenio, and Marcus Größer. Partial order reduction for probabilistic branching time. *Electr. Notes Theor. Comput. Sci.*, 153(2):97–116, 2006. 3.4

[11] Christel Baier and Joost P. Katoen. *Principles of Model Checking*. The MIT Press, May 2008. 1, 2.4, 3.1.1

[12] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A Decade of Software Model Checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011. 2.4

[13] Paolo Ballarini and Alice Miller. Model Checking Medium Access Control for Sensor Networks. In *ISoLA*, pages 255–262, 2006. 1, 3.1.4

[14] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *SEFM*, pages 3–12, 2006. 3.2

[15] A. Basu, L. Mounier, M. Poulhiès, J. Pulou, and J. Sifakis. Using BIP for Modeling and Verification of Networked Systems – A Case Study on TinyOS-based Networks. In *NCA*, pages 257–260, 2007. 3.2

[16] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. DMS®: Program Transformations for Practical Scalable Software Evolution. In *ICSE*, pages 625–634, 2004. 3.4

[17] Johan Bengtsson, Bengt Jonsson, Johan Lilius, and Wang Yi. Partial Order Reductions for Timed Systems. In 9^{th} *International Conference on Concurrency Theory*, volume 1466 of *Lecture Notes in Computer Science*, pages 485–500. Springer-Verlag, 1998. 3.4

[18] Johan Bengtsson, Kim G. Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Uppaal — a Tool Suite for Automatic Verification of Real–Time Systems. In *Proc. of Workshop on Verification and Control of Hybrid Systems III*, number 1066 in Lecture Notes in Computer Science, pages 232–243. Springer–Verlag, October 1995. 3.1.3

[19] Jan A. Bergstra and Jan Willem Klop. $\mathrm{ACT}_{\mathrm{tau}}$: A Universal Axiom System for Process Specification. In *Algebraic Methods*, pages 447–463, 1987. 3.1.1

[20] Tommaso Bolognesi and Ed Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks*, 14:25–59, 1987. 1

[21] M. Bozga, S. Graf, and L. Mounier. IF-2.0: A Validation Environment for Component-Based Real-Time Systems. In *CAV '02*, pages 343–348, London, UK, 2002. Springer-Verlag. 3.2

[22] Lubos Brim, Ivana Cerná, Pavel Moravec, and Jirí Simsa. Distributed Partial Order Reduction of State Spaces. *Electr. Notes Theor. Comput. Sci.*, 128(3):63–74, 2005. 3.4

[23] Philippa J. Broadfoot and A. W. Roscoe. Tutorial on FDR and Its Applications. In *SPIN*, page 322, 2000. 2.4

[24] Doina Bucur. Intelligible TinyOS Sensor Systems: Explanations for Embedded Software. In *CONTEXT*, pages 54–66, 2011. 1, 8

[25] Doina Bucur and Marta Z. Kwiatkowska. Bug-Free Sensors: The Automatic Verification of Context-Aware TinyOS Applications. In *AMI*, pages 101–105, Salzburg, Austria, 2009. 1, 3.3

[26] Doina Bucur and Marta Z. Kwiatkowska. Software verification for TinyOS. In *IPSN*, pages 400–401, 2010. 1, 3.3

[27] Doina Bucur and Marta Z. Kwiatkowska. On software verification for sensor nodes. *Journal of Systems and Software*, 84(10):1693–1707, 2011. 1, 3.3, 3.4

[28] Ricky W. Butler. What is formal methods? Online reference: http://shemesh.larc.nasa.gov/fm/fm-what.html [6 August 2001]. D

[29] Chunqing Chen, Jun Sun, Yang Liu, Jin Song Dong, and Manchun Zheng. Formal modeling and validation of stateflow diagrams. *The International Journal on Software Tools for Technology Transfer (STTT)*, 2012. 1.2, 7.1

[30] Gabriel Ciobanu and Manchun Zheng. Automatic Analysis of TiMo Systems in PAT. In *ICECCS*, 2013. 1.2

[31] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. 1, 3.1.4

[32] Edmund M. Clarke, Orna Grumberg, and Somesh Jha. Veryfying Parameterized Networks using Abstraction and Regular Languages. In *CONCUR*, pages 395–407, 1995. 8

[33] Edmund M. Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State Space Reduction Using Partial Order Techniques. *STTT*, 2(3):279–287, 1999. 3.4

[34] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001. 1.1, 3.4, 6, 6.4

[35] Ariel Cohen and Kedar S. Namjoshi. Local proofs for linear-time properties of concurrent programs. In *CAV*, pages 149–161, 2008. 8

[36] Ariel Cohen and Kedar S. Namjoshi. Local proofs for global safety properties. *Formal Methods in System Design*, 34(2):104–125, 2009. 8

[37] Ariel Cohen, Kedar S. Namjoshi, and Yaniv Sa'ar. A dash of fairness for compositional reasoning. In *CAV*, pages 543–557, 2010. 8

[38] S. Coleri, M. Ergen, and T. J. Koo. Lifetime analysis of a sensor network with hybrid automata modelling. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 98–104, New York, NY, USA, 2002. ACM. 1, 3.2

[39] Véronique Cortier, Jan Degrieck, and Stéphanie Delaune. Analysing Routing Protocols: Four Nodes Topologies Are Sufficient. In *POST*, pages 30–50, 2012. 8

[40] David E. Culler, Jason Hill, Philip Buonadonna, Robert Szewczyk, and Alec Woo. A Network-Centric Approach to Embedded Software for Tiny Devices. In *EMSOFT*, pages 114–130, 2001. 2.1

[41] Pedro R. D'Argenio and Peter Niebert. Partial Order Reduction on Concurrent Probabilistic Programs. In *QEST*, pages 240–249, 2004. 3.4

[42] Jim Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993. C

[43] Jim Davies and Steve Schneider. A Brief History of Timed CSP. *Theor. Comput. Sci.*, 138(2):243–271, 1995. D

[44] Jin Song Dong and Jun Sun. Towards Expressive Specification and Efficient Model Checking. In Wei-Ngan Chin and Shengchao Qin, editors, *Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'09)*, page 9. IEEE Computer Society, 2009. 7.1

[45] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the 1st IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, pages 455–462, Tampa, Florida, USA, 2004. 3.3

[46] Matthew B. Dwyer and Lori A. Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. In *SIGSOFT FSE*, pages 62–75, 1994. 3.4

[47] E. Allen Emerson, Somesh Jha, and Doron Peled. Combining Partial Order and Symmetry Reductions. In *TACAS*, pages 19–34, 1997. 3.4

[48] Azadeh Farzan and José Meseguer. Partial Order Reduction for Rewriting Semantics of Programming Languages. *Electr. Notes Theor. Comput. Sci.*, 176(4):61–78, 2007. 3.4

[49] Ansgar Fehnker, Lodewijk van Hoesel, and Angelika Mader. Modelling and Verification of the LMAC Protocol for Wireless Sensor Networks. In *IFM*, pages 253–272, 2007. 8

[50] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-order Reduction for Model Checking Software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005. 1.1, 3.4

[51] Wan Fokkink. *Introduction to Process Algebra*. Texts in theoretical computer science. Springer, 2000. 3.1.1

[52] D. Gay, P. Levis, and D. E. Culler. Software design patterns for TinyOS. *ACM Trans. Embedded Comput. Syst.*, 6(2), 2007. 1.1, 2.1

[53] 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. 1, 1.1, 2.2, 2.3, D

[54] David Gay, Philip Levis, and David Culler. Software Design Patterns for TinyOS. In *LCTES*, pages 40–49, Illinois, USA, 2005. 1.1, 2.1

[55] David Gay, Philip Levis, David Culler, and Eric Brewer. nesC 1.3 Language Reference Manual, 2009. 2.2, A

[56] Sergio Giro, Pedro R. D'Argenio, and Luis María Ferrer Fioriti. Partial Order Reduction for Probabilistic Systems: A Revision for Distributed Schedulers. In *CONCUR*, pages 338–353, 2009. 3.4

[57] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection Tree Protocol. In *SenSys*, pages 1–14, 2009. 2.1

[58] Patrice Godefroid. Using Partial Orders to Improve Automatic Verification Methods. In *CAV*, pages 176–185, 1990. 3.4

[59] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. D

[60] Patrice Godefroid, Doron Peled, and Mark G. Staskauskas. Using Partial-Order Methods in the Formal Validation of Industrial Concurrent Programs. In *ISSTA*, pages 261–269, 1996. 3.4

[61] Patrice Godefroid and Pierre Wolper. Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties. *Formal Methods in System Design*, 2(2):149–164, 1993. 1.1, 6.2

[62] J. Green, S. Bhattacharyya, and B. Panja. Real-Time Logic Verification of a Wireless Sensor Network. *World Congress Computer Science and Information Engineering*, 3:269–273, 2009. 3.1.3

[63] J. Green, S. Bhattacharyya, and Biswajit Panja. Real-Time Logic Verification of a Wireless Sensor Network. In *CSIE*, pages 269–273, Los Angeles, USA, 2009. 1

[64] Rachid Guerraoui and Maysam Yabandeh. Model Checking a Networked System without the Network. In *NSDI*, 2011. 8

[65] Guy Gueta, Cormac Flanagan, Eran Yahav, and Mooly Sagiv. Cartesian Partial-Order Reduction. In *SPIN*, pages 95–112, 2007. 1.1, 3.4, 6, 6.2, D

[66] Y. Hanna. SLEDE: lightweight verification of sensor network security protocol implementations. In *ESEC*, pages 591–594, Dubrovnik, Croatia, 2007. 1, 3.3

[67] Y. Hanna and H. Rajan. SLEDE: event-based specification of sensor network security protocols. *ACM SIGSOFT Software Engineering Notes*, 31(6):1–2, 2006. 3.3

[68] Y. Hanna and H. Rajan. SLEDE: Framework for automatic verification of sensor network security protocol implementations. In *ICSE*, pages 427–428, Vancouver, Canada, 2009. 3.3, 8

[69] 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. 1, 1.1, 3.3

[70] H. Hansson and B. Jonsson. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing*, 6:102–111, 1994. 3.1.4

[71] Jianye Hao, Songzheng Song, Yang Liu, Jun Sun, Lin Gui, Jin Song Dong, and Ho fung Leung. Probabilistic Model Checking Multi-agent Behaviors in Dispersion Games Using Counter Abstraction. In *PRIMA*, pages 16–30, 2012. 7.1

[72] Klaus Havelund, Mike Lowry, and John Penix. Formal analysis of a space-craft controller using spin. *IEEE Trans. Softw. Eng.*, 27:749–765, August 2001. 1, 2.4

[73] Klaus Havelund and Thomas Pressburger. Model Checking JAVA Programs using JAVA PathFinder. *STTT*, 2(4):366–381, 2000. 2.4

[74] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985. 1, 3.1.1, 5.2.1, C, D

[75] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual.* Addison-Wesley, 2004. 1, 3.1.1

[76] Gerard J. Holzmann and Rajeev Joshi. Model-Driven Software Verification. In *PASTE*, pages 80–89, 2001. 2.4

[77] Vineet Kahlon, Chao Wang, and Aarti Gupta. Monotonic Partial Order Reduction: An Optimal Symbolic Partial Order Reduction Technique. In *CAV*, pages 398–413, 2009. 3.4

[78] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, and Armaghan Naik. Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation. In *CAV*, pages 414–429, Grenoble, France, 2009. 1, 2.4

[79] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Razvan Musaloiu-Elefteri, Philip Levis, Andreas Terzis, and Ramesh Govindan. TOSThreads: thread-safe and non-invasive preemption in TinyOS. In *SenSys*, pages 127–140, 2009. 8

[80] N. Kothari, T. D. Millstein, and R. Govindan. Deriving State Machines from TinyOS Programs Using Symbolic Execution. In *IPSN*, pages 271–282, 2008. 1, 3.3

[81] Sudipta Kundu, Malay K. Ganai, and Rajesh Gupta. Partial order reduction for scalable testing of systemC TLM designs. In *DAC*, pages 936–941, 2008. 3.4

[82] Robert P. Kurshan, Vladimir Levin, Marius Minea, Doron Peled, and Hüsnü Yenigün. Static Partial Order Reduction. In *TACAS*, pages 345–357, 1998. 3.4

[83] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Model Checking for Performance and Reliability Analysis. *ACM SIGMETRICS Performance Evaluation Review*, 36(4):40–45, 2009. 3.1.4

[84] Á. Lédeczi, A. Bakay, M. Maroti, P. Völgyesi, G. Nordstrom, J. Sprinkle, and G. Karsai. Composing Domain-Specific Design Environments. *IEEE Computer*, 34(11):44–51, 2001. 3.2, 3.2

[85] P. Levis and D. Gay. *TinyOS Programming.* Cambridge University Press, 1 edition, 2009. 2.1, 2.1, 2.2, 2.3

[86] 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. 1, 3.3

[87] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An operating system for sensor networks. In *Ambient Intelligence.* Springer Verlag, 2004. 1, 2.1

[88] Philip Levis, Neil Patel, David E. Culler, and Scott Shenker. Trickle: A Self-Regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *NSDI*, pages 15–28, California, USA, 2004. 1.1, 2.1, 1, 5.6, 6.5.2

[89] Peng Li and John Regehr. T-Check: bug finding for sensor networks. In *IPSN*, pages 174–185, Stockholm, Sweden, 2010. (document), 1, 1.1, 1.1, 1.3, 3.3, 3.4, 6, 6.5, 7, 8

[90] Yi Li, Jing Sun, Jin Song Dong, Yang Liu, and Jun Sun. Translating PDDL into CSP# - The PAT Approach. In *ICECCS*, pages 240–249, 2012. 7.1

[91] Shang-Wei Lin, Étienne André, Jin Song Dong, Jun Sun, and Yang Liu. An Efficient Algorithm for Learning Event-Recording Automata. In Tevfik Bultan and Pao-Ann Hsiung, editors, *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA'11)*, volume 6996 of *Lecture Notes in Computer Science*, pages 463–472, Taipei, Taiwan, October 2011. Springer. 7.1

[92] Shang-Wei Lin, Yang Liu, Jun Sun, Jin Song Dong, and Étienne André. Automatic Compositional Verification of Timed Systems. In *FM*, pages 272–276, 2012. 7.1

[93] Yan Liu, Xian Zhang, Jin Song Dong, Yang Liu, Jun Sun, Jit Biswas, and Mounir Mokhtari. Formal Analysis of Pervasive Computing Systems. In *ICECCS*, pages 169–178, 2012. 7.1

[94] Yang Liu, Wei Chen, Yanghong A. Liu, Shao Jie Zhang, Jun Sun, and Jin Song Dong. Verifying Linearizability via Optimized Refinement Checking. *IEEE Transactions on Software Engineering*, 2013. 7.1

[95] Yang Liu, Wei Chen, Yanhong A. Liu, and Jun Sun. Model Checking Linearizability via Refinement. In Ana Cavalcanti and Dennis Dams, editors, *Proceedings of the Second World Congress on Formal Methods (FM'09)*, volume 5850 of *Lecture Notes in Computer Science*, pages 321–337. Springer, 2009. 7.1

[96] Yang Liu, Jun Pang, Jun Sun, and Jianhua Zhao. Verification of Population Ring Protocols in PAT. In Wei-Ngan Chin and Shengchao Qin, editors, *Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'09)*, pages 81–89. IEEE Computer Society, 2009. 7.1

[97] Yang Liu, Jun Sun, and Jin Song Dong. An Analyzer for Extended Compositional Process Algebras. In *ICSE Companion*, pages 919–920. ACM, 2008. 1.1, 7, 7.1

[98] Yang Liu, Jun Sun, and Jin Song Dong. Scalable Multi-core Model Checking Fairness Enhanced Systems. In Karin Breitman and Ana Cavalcanti, editors, *Proceedings of the 11th IEEEInternational Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *Lecture Notes in Computer Science*, pages 426–445. Springer, 2009. 7.1

[99] Yang Liu, Jun Sun, and Jin Song Dong. Developing Model Checkers Using PAT. In Ahmed Bouajjani and Wei-Ngan Chin, editors, *ATVA*, pages 371–377, Singapore, 2010. Springer. 1.1, 7, 7.1

[100] Yang Liu, Jun Sun, and Jin Song Dong. PAT 3: An Extensible Architecture for Building Multi-domain Model Checkers. In *ISSRE*, Hiroshima, Japan, 2011. 7.1

[101] L. Lopes, F. Martins, M. S. Silva, and J. Barros. A Formal Model for Programming Wireless Sensor Networks. *Informal Publication*, 2007. 3.1.2

[102] Bas Luttik and Nikola Trcka. Stuttering Congruence for *chi*. In *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings (SPIN'05)*, pages 185–199, 2005. 6.4

[103] Anh Tuan Luu, Jun Sun, Yang Liu, Jin Song Dong, Xiaohong Li, and Quan Thanh Tho. SeVe: automatic tool for verification of security protocols. *Frontiers of Computer Science in China*, 6(1):57–75, 2012. 7.1

[104] L. Mandel and M. Pouzet. ReactiveML, a Reactive Extension to ML. In *Proceedings of 7th ACM SIGPLAN International conference on Principles and Practice of Declarative Programming (PPDP'05)*, Lisbon, Portugal, July 2005. 3.1.1

[105] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems:Specification*. Springer-Verlag, 1992. 1, 1.1

[106] F. Maraninchi, L. Samper, K. Baradon, and A. Vasseur. Lustre as a System Modeling Language: Lussensor, a Case-Study with Sensor Networks. *Electron. Notes Theor. Comput. Sci.*, 203(4):95–110, 2008. 1, 3.1.1

[107] A. I. McInnes. Using CSP to Model and Analyze TinyOS Applications. In *ECBS*, pages 79–88, California, USA, 2009. 1, 3.2

[108] Volker Menrad, Miguel Garcia, and Sibylle Schupp. Improving TinyOS Developer Productivity with State Charts. In *SOMSED*, 2009. 3.3

[109] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980. 3.1.1

[110] Marius Minea. Partial Order Reduction for Model Checking of Timed Automata. In *CONCUR*, pages 431–446, 1999. 3.4

[111] Luca Mottola, Thiemo Voigt, Fredrik Osterlind, Joakim Eriksson, Luciano Baresi, and Carlo Ghezzi. Anquiro: Enabling Efficient Static Verification of Sensor Network Software. In *SESENA*, pages 32–37, 2010. 1, 1.1, 1.1, 3.3

[112] Ratan Nalumasu and Ganesh Gopalakrishnan. New Partial Order Reduction Algorithm for Concurrent System Verification. In *Proc of IFIP*. Chapman & Hall, 1997. 3.4

[113] Kedar S. Namjoshi. Symmetry and completeness in the analysis of parameterized systems. In *VMCAI*, pages 299–313, 2007. 8

[114] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. Data Flow Analysis for Checking Properties of Concurrent Java Programs. In *ICSE*, pages 399–410, 1999. 3.4

[115] Gleb Naumovich, Lori A. Clarke, and Jamieson M. Cobleigh. Using partial order techniques to improve performance of data flow analysis based verification. In *In Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 57–65, 1999. 3.4

[116] Nguyet T. M. Nguyen and Mary Lou Soffa. Program representations for testing wireless sensor network applications. In *DOSTA*, pages 20–26, 2007. 3.3

[117] Truong Khanh Nguyen, Jun Sun, Yang Liu, and Jin Song Dong. A model checking framework for hierarchical systems. In *ASE*, pages 633–636, 2011. 7.1

[118] Truong Khanh Nguyen, Jun Sun, Yang Liu, and Jin Song Dong. Symbolic Model-Checking of Stateful Timed CSP Using BDD and Digitization. In *ICFEM*, pages 398–413, 2012. 7.1

[119] Truong Khanh Nguyen, Jun Sun, Yang Liu, Jin Song Dong, and Yan Liu. Improved BDD-Based Discrete Analysis of Timed Systems. In *FM*, pages 326–340, 2012. 7.1

[120] Vladimir A. Oleshchuk. Modeling, specification and verification of ad-hoc sensor networks using SPIN. *Computer Standards & Interfaces*, 28(2):159–165, 2005. 1, 3.1.1

[121] P. C. Ölveczky and S. Thorvaldsen. Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in Real-Time Maude. *Theor. Comput. Sci.*, 410(2-3):254–280, 2009. 1, 3.1.3

[122] Peter Csaba Ölveczky and Stian Thorvaldsen. Formal Modeling and Analysis of the OGDC Wireless Sensor Network Algorithm in Real-Time Maude. In *FMOODS*, pages 122–140, 2007. 1, 3.1.3

[123] Robert Palmer, Ganesh Gopalakrishnan, and Robert M. Kirby. Semantics driven dynamic partial-order reduction of MPI-based parallel programs. In *PADTAD*, pages 43–53, 2007. 3.4

[124] Doron Peled. All from One, One for All: on Model Checking Using Representatives. In *CAV*, pages 409–423, 1993. D

[125] Doron Peled. Combining Partial Order Reductions with On-the-Fly Model-Checking. *Formal Methods in System Design*, 8(1):39–64, 1996. 3.4

[126] Doron Peled. Partial Order Reduction: Model-Checking Using Representatives. In *MFCS*, pages 93–112, 1996. 3.4

[127] Doron Peled. Ten Years of Partial Order Reduction. In *CAV*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28, Vancouver, Canada, 1998. Springer. 3.4

[128] Luiz Felipe Perrone and David M. Nicol. A scalable simulator for TinyOS applications. In *WSC*, 2002. 1

[129] Amir Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57, 1977. (document), 2.2

[130] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC / SIGSOFT FSE*, pages 267–276, 2003. 3.3

[131] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: A Flexible Framework for Creating Software Model Checkers. In *TAIC PART*, pages 3–22, 2006. 3.3

[132] N. S. Rosa and P. R. F. Cunha. Behavioural Specification of Wireless Sensor Network Applications. In *GIIS*, pages 66–72, 2007. 1, 3.1.1, 3.2

[133] L. Samper, F. Maraninchi, L. Mounier, and L. Mandel. GLONEMO: Global and Accurate Formal Models for the Analysis of Ad-Hoc Sensor Networks. In *InterSense: First International Conference on Integrated Internet Ad hoc and Sensor Networks*, Nice, France, May 2006. IEEE. 1, 3.1.1

[134] Steve Schneider. An Operational Semantics for Timed CSP. *Inf. Comput.*, 116(2):193–213, 1995. D

[135] Ling Shi and Yan Liu. Modeling and Verification of Transmission Protocols: A Case Study on CSMA/CD Protocol. *Secure Software Integration and Reliability Improvement Companion, IEEE International Conference on*, 0:143–149, 2010. 7.1

[136] Ling Shi, Yang Liu, Jun Sun, Jin Song Dong, and Gustavo Carvalho. An Analytical and Experimental Comparison of CSP Extensions and Tools. In *ICFEM*, 2012. 7.1

[137] Songzheng Song. An Efficient Method of Probabilistic Model Checking. *Secure Software Integration and Reliability Improvement Companion, IEEE International Conference on*, 0:24–25, 2010. 7.1

[138] Songzheng Song, Jianye Hao, Yang Liu, Jun Sun, Ho-Fung Leung, and Jin Song Dong. Analyzing multi-agent systems with probabilistic model checking approach. In *ICSE*, pages 1337–1340, 2012. 7.1

[139] Songzheng Song, Jun Sun, Yang Liu, and Jin Song Dong. A Model Checker for Hierarchical Probabilistic Real-Time Systems. In *CAV*, pages 705–711, 2012. 7.1

[140] Jun Sun, Yang Liu, and Bin Cheng. Model Checking a Model Checker: A Code Contract Combined Approach. In Jin Song Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering - 12^{th} International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 518–533. Springer, 2010. 7.1

[141] Jun Sun, Yang Liu, and Jin Song Dong. Model Checking CSP Revisited: Introducing a Process Analysis Toolkit. In *Proceedings of the 3^{rd} International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*, volume 17 of *Communications in Computer and Information Science*, pages 307–322. Springer, 2008. 7.1

[142] Jun Sun, Yang Liu, Jin Song Dong, and Chunqing Chen. Integrating Specification and Programs for System Modeling and Verification. In Wei-Ngan Chin and Shengchao Qin, editors, *Proceedings of the 3^{rd} IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE'09)*, pages 127–135. IEEE Computer Society, 2009. 7.1

[143] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP. *ACM Trans. Softw. Eng. Methodol.*, 2012. 1.1, 4, 4.1, 4.3, 7.1, C

[144] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 709–714, 2009. 1.1, 4.3, 5.5, 7, 7.1, 7.2.4

[145] Jun Sun, Yang Liu, Jin Song Dong, Geguang Pu, and Tian Huat Tan. Model-based Methods for Linking Web Service Choreography and Orchestration. In *APSEC 2010*, pages 166 – 175, 2010. 7.1

[146] Jun Sun, Yang Liu, Jin Song Dong, and Jing Sun. Bounded Model Checking of Compositional Processes. In *Proceedings of the 2nd IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 23–30. IEEE Computer Society, 2008. 7.1

[147] Jun Sun, Yang Liu, Jin Song Dong, and Hai H. Wang. Specifying and verifying event-based fairness enhanced systems. In *Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM 2008)*, volume 5256 of *Lecture Notes in Computer Science*, pages 318–337. Springer, Oct 2008. 7.1

[148] Jun Sun, Yang Liu, Jin Song Dong, and Xian Zhang. Verifying Stateful Timed CSP Using Implicit Clocks and Zone Abstraction. In Karin Breitman and Ana Cavalcanti, editors, *Proceedings of the 11th IEEEInternational Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *Lecture Notes in Computer Science*, pages 581–600. Springer, 2009. 4, 4.2, 4.3, 7.1

[149] Jun Sun, Yang Liu, Abhik Roychoudhury, Shanshan Liu, and Jin Song Dong. Fair Model Checking with Process Counter Abstraction. In *FM*, pages 123–139, 2009. 5.5, 7.1, 7.2.4

[150] Jun Sun, Yang Liu, Songzheng Song, Jin Song Dong, and Xiaohong Li. PRTS: An Approach for Model Checking Probabilistic Real-Time Hierarchical Systems. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 147–162. Springer Berlin / Heidelberg, 2011. 7.1

[151] Jun Sun, Songzheng Song, and Yang Liu. Model Checking Hierarchical Probabilistic Systems. In Jin Song Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2010. 7.1

[152] Tian Huat Tan. Towards Verification of a Service Orchestration Language. *Secure Software Integration and Reliability Improvement Companion, IEEE International Conference on*, 0:36–37, 2010. 7.1

[153] Tian Huat Tan, Yang Liu, Jun Sun, and Jin Song Dong. Verification of Orchestration Systems Using Compositional Partial Order Reduction. In Shengchao Qin and Zongyan Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 98–114. Springer Berlin / Heidelberg, 2011. 7.1

[154] Luu Anh Tuan. Modeling and Verifying Security Protocols Using PAT Approach. *Secure Software Integration and Reliability Improvement Companion, IEEE International Conference on*, 0:157–164, 2010. 7.1

[155] Luu Anh Tuan, Man Chun Zheng, and Quan Thanh Tho. Modeling and verification of safety critical systems: A case study on pacemaker. *Secure System Integration and Reliability Improvement*, 0:23–32, 2010. 1.2, 7.1

[156] Antti Valmari. Stubborn sets for reduced state space generation. In *Applications and Theory of Petri Nets*, pages 491–515, 1989. D

[157] Willem Visser, Klaus Havelund, Guillaume P. Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003. 3.4

[158] P. Völgyesi, M. Maróti, S. Dóra, E. Osses, and Á. Lédeczi. Software Composition and Verification for Sensor Networks. *Sci. Comput. Program.*, 56(1-2):191–210, 2004. 1, 3.2, 3.2

[159] P. Völgyesi, M. Maróti, S. Dóra, E. Osses, Á. Lédeczi, and T. Paka. Embedded Software Composition and Verification. Technical Report ISIS-04-503, Vanderbilt University, 2004. 1, 3.2

[160] Ting Wang, Songzheng Song, Jun Sun, Yang Liu, Jin Song Dong, Xinyu Wang, and Shanping Li. More Anti-chain Based Refinement Checking. In *ICFEM*, pages 364–380, 2012. 7.1

[161] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003. 3.2

[162] Frank Werner and David Faragó. Correctness of Sensor Network Applications by Software Bounded Model Checking. In *FMICS*, pages 115–131, 2010. 1, 1.1, 3.3

[163] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed Dynamic Partial Order Reduction Based Verification of Threaded Software. In *SPIN*, pages 58–75, 2007. 3.4

[164] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient Stateful Dynamic Partial Order Reduction. In *SPIN*, volume 5156 of *LNCS*, pages 288–305. Springer, 2008. 1.1, 3.4

[165] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Distributed dynamic partial order reduction. *STTT*, 12(2):113–122, 2010. 3.4

[166] Shao Jie Zhang and Yang Liu. An Automatic Approach to Model Checking UML State Machines. *Secure Software Integration and Reliability Improvement Companion, IEEE International Conference on*, 0:1–6, 2010. 7.1

[167] Shao Jie Zhang and Yang Liu. Model Checking a Lazy Concurrent List-Based Set Algorithm. In *Proceedings of the 4^{th} International Conference on Secure Software Integration and Reliability Improvement*, SSIRI '10, pages 43–52, Washington, DC, USA, 2010. IEEE Computer Society. 7.1

[168] Shao Jie Zhang, Yang Liu, Jun Sun, Jin Song Dong, Wei Chen, and Yanhong A. Liu. Formal Verification of Scalable NonZero Indicators. In *Proceedings of the 21^{st} International Conference on Software Engineering & Knowledge Engineering (SEKE'2009)*, pages 406–411. Knowledge Systems Institute Graduate School, 2009. 7.1

[169] Shao Jie Zhang, Jun Sun, Jun Pang, Yang Liu, and Jin Song Dong. On Combining State Space Reductions with Global Fairness Assumptions. In Michael Butler and Wolfram Schulte, editors, *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 432–447. Springer Berlin / Heidelberg, 2011. 7.1

[170] Man Chun Zheng. An Automatic Approach to Verify Sensor Network Systems. *Secure Software Integration and Reliability Improvement Companion, IEEE International Conference on*, 0:7–12, 2010. 1.2

[171] Manchun Zheng, David Sanán, Jun Sun, Yang Liu, Jin Song Dong, and Yu Gu. State Space Reduction for Sensor Networks using Two-level Partial Order Reduction. In *14^{th} International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2013. 1.2

[172] Manchun Zheng, Jun Sun, Yang Liu, Jin Song Dong, and Yu Gu. Towards a model checker for nesc and wireless sensor networks. In *ICFEM*, pages 372–387, 2011. 1, 1.2, 6

[173] Manchun Zheng, Jun Sun, David Sanán, Yang Liu, Jin Song Dong, and Yu Gu. Towards bug-free implementation for wireless sensor networks. In *SenSys*, pages 407–408, 2011. 1.2

[174] Huiquan Zhu. Model Checking C# Code: A Translation Approach. *Secure Software Integration and Reliability Improvement Companion, IEEE International Conference on*, 0:30–31, 2010. 7.1

Appendix A

NesC Language Reference

This chapter presents the syntax of the NesC language, based on the NesC 1.3 Language Reference Manual [55].

A.1 Interface Definition

Interface definitions have the following syntax:

interface-definition :
interface *identifier type-parameters*$_{opt}${*declaration-list*}

Interface definitions have a name (identifier) with global scope. The $type - parameters$ is a list of optional C type parameters for this interface definition:

type-parameters :
$<$ *type-parameter-list* $>$

type-parameter-list :
identifier
type-parameter-list, *identifier*

An interface definition with type parameters is called a generic interface definition. The *declaration-list* of an interface definition specifies a set of commands and events. It must consist of function declarations with the **command** or **event** storage class:

storage-class-specifier : *also one of*
command event async

An interface type is specified by giving the name of an interface definition and, for generic

interface definitions, any required type arguments:

interface-type :
    interface *identifier* *type-arguments*$_{opt}$

type-arguments :
    < *type-argument-list* >

type-argument-list :
    type-name
    type-argument-list, *type-name*

There must be as many types in type-arguments as there are parameters in the interface definitionąś type parameter list. Type arguments can not be incomplete or of function or array type. Two interface types are the same if they refer to the same interface definition and their corresponding type arguments (if any) are of the same C type. Example interface types are interface SendMsg and interface Queue<int>.

A.2 Component Specification

The first part of a componentąś definition is its specification, a declaration of provided or used specification elements, where each element is an interface, a bare command or event or a declaration:

component-specification :
    {*uses-provides-list*}

uses-provides-list :
    uses-provides
    uses-provides-list *uses-provides*

uses-provides :
    uses *specification-element-list*
    provides *specification-element-list*

specification-element-list :
    specification-element
    {*specification-elements*}

specification-elements :
    specification-element
    specification-elements *specification-element*

An interface declaration has an interface type and an optional name:

specification-element :
    interface-type *instance-name*$_{opt}$ *instance-parameters*$_{opt}$
    ...

instance-name :
    as identifier

instance-parameters :
    [*parameter-type-list*]

A specification can contain independent interfaces of the same interface type, e.g.,

provides interface X as X1;
uses interface X as X2;

Commands or events can be included directly as specification elements by including a standard C function declaration with **command** or **event** as its storage class specifier:

specification-element :
    declaration
    ...

storage-class-specifier : *also one of*
    command event async

It is a compile-time error if the declaration is not a function declaration with the command or event storage class. As in interfaces, async indicates that the command or event can be called from an interrupt handler.

As with interface declarations, bare commands (bare events) can have instance parameters; these are placed before the functionąŕs regular parameter list, e.g., command void send[uint8 t id](int x):

direct-declarator : *also*
    direct-declarator *instance-parameters*(*parameter-type-list*)
    ...

If instance parameters are present, the declaration specifies a bare, parameterised command (bare, parameterised event). Note that instance parameters are not allowed on commands or events inside interface definitions.

A component specification can also include regular declarations (these belong to the

specification scope):

uses − provides : *also*
    declaration

A.3 Component Definition

A nesC component definition has a name, optional arguments, a specification and an implementation:

component :
    comp-kind identifier comp-parameters$_{opt}$ *component-specification implementation*$_{opt}$

comp-kind :
    module
    configuration
    component
    generic module
    generic configuration

implementation :
    module-implementation
    configuration-implementation

Generic component parameter lists are similar to function parameter lists, but allow for type parameters by (re)using the **typedef** keyword:

comp-parameters :
    (*component-parameter-list*)

component-parameter-list :
    component-parameter
    component-parameter-list, *component-parameter*

component-parameter :
    parameter-declaration
    typedef *identifier*

A.4 Module Implementation

Modules implement a component specification with C code:

module-implementation :
    implementation{*translation-unit*}

where translation-unit is a list of C declarations and definitions. The top-level declarations of the moduleąŕs translation-unit belong to the moduleąŕs implementation scope. These declarations have indefinite extent and can be: any standard C declaration or definition, a task declaration or definition (placed in the object name space), a command or event implementation.

These command and event implementations are specified with the following C syntax extensions:

storage-class-specifier : *also one of*
    command event async

declaration-specifiers : *also*
    default declaration-specifiers

direct-declarator : *also*
    identifier.*identifier*
    direct-declarator interface-parameters(*parameter-type-list*)

The following extensions to C syntax are used to call events and signal commands:

postfix-expression :
    postfix-expression[*argument-expression-list*]
    $call\text{-}kind_{opt}$ *primary*($argument\text{-}expression\text{-}list_{opt}$)
...

call-kind : *one of*
    call signal post

A module can specify a default implementation for a used command or event α (a compile-time error occurs if a default implementation is supplied for a provided command or event). Default implementations are executed when α is not connected to any command or event implementation. A default command or event is defined by prefixing a command or event implementation with the **default** keyword:

declaration-specifiers : *also*
    default *declaration-specifiers*

A task is an independent locus of control defined by a function of storage class task returning void and with no arguments: task void myTask() { ... }. A task can also have a

forward declaration, e.g., task void myTask();.

storage-class-specifier : *also one of*
task

call-kind : *also one of*
post

Atomic statements:

atomic-stmt :
atomic *statement*

guarantee that the statement is executed "as-if" no other computation occurred simultaneously, and furthermore any values stored inside an atomic statement are visible inside all subsequent atomic statements.

A.5 Configuration Implementation

Configurations implement a component specification by selecting regular components or instantiating generic components, and then connecting (ąřwiringąś) these components together. The implementation section of a configuration consists of a list of configuration elements:

configuration-implementation :
implementation {*configuration-element-list*$_{opt}$}

configuration-element-list :
configuration-element
configuration-element-list configuration-element

configuration-element :
components
connection
declaration

A components element specifies the components that are used to build this configuration, a connection specifies a single wiring statement, and a declaration can declare a **typedef** or tagged type (other C declarations are compile-time errors). A configuration composed of wiring statements connects two sets of specification elements:

- C's specification elements. In this section, we refer to these as *external* specification elements.

- The specification elements of the components referred to instantiated in C. We refer to these as internal specification elements.

A components elements specifies some components used to build this configuration. These can be:

- A non-generic component X. Non-generic components are implicitly instantiated, references to X in different configurations all refer to the same component.
- An instantiation of a generic component Y . Instantiations of Y in different configurations, or multiple instantiations in the same configuration represent different components.

The syntax of components is as follows:

components :
    components *component-line* **;**

component-line :
    component-ref instance-name$_{opt}$
    component-line **,** *component-ref instance-name*$_{opt}$

instance-name :
    as *identifier*

component-ref :
    identifier
    new *identifier*(*component-argument-list*)

component-argument-list :
    component-argument
    component-argument-list **,** *component-argument*

component-argument :
    expression
    type-name

Wiring is used to connect specification elements (interfaces, commands, events) together.

connection :
    endpoint = *endpoint*
    endpoint → *endpoint*
    endpoint ← *endpoint*

endpoint :
    identifier-path
    identifier-path [*argument-expression-list*]
identifier-path :
    identifier
    identifier-path **.** *identifier*

There are three wiring statements in nesC:

- $endpoint_1 = endpoint_2$ (equate wires): Any connection involving an external specification element. These effectively make two specification elements equivalent. Let S_1 be the specification element of $endpoint_1$ and S_2 that of $endpoint_2$. One of the following two conditions must hold or a compile-time error occurs:
  - S_1 is internal, S_2 is external (or vice-versa) and S_1 and S_2 are both provided or both used,
  - S_1 and S_2 are both external and one is provided and the other used.
- $endpoint_1 \rightarrow endpoint_2$ (link wires): A connection between two internal specification elements. Link wires always connect a used specification element specified by $endpoint_1$ to a provided one specified by $endpoint_2$. If these two conditions do not hold, a compile-time error occurs.
- $endpoint_1 \leftarrow endpoint_2$ is equivalent to $endpoint_2 \rightarrow endpoint_1$.

A configuration can refer to the **typedefs** and enum constants of the components that it includes. To support this, the syntax for referring to **typedefs** is extended as follows:

typedef-name : *also one of*
    identifier **.** *identifier*

Appendix B

Syntax of Logical Formulas

In the following, we present the BNF syntax of logical formulas which are used in the assertion annotation language of NesC@PAT.

```
mathExpr : conditionalAndExpr (|| conditionalAndExpr)*

conditionalAndExpr : bitwiseOrExpr (&&bitwiseOrExpr)*

bitwiseOrExpr : bitwiseXorExpr (| bitwiseXorExpr)*

bitwiseXorExpr : bitwiseAndExpr (?bitwiseAndExpr)*

bitwiseAndExpr : equalityExpr (&equalityExpr)*

equalityExpr : relationalExpr ((== | !=)relationalExpr)*

relationalExpr : bitwiseShiftExpr ((< | > | <= | >=) bitwiseShiftExpr)*

bitwiseShiftExpr : additiveExpr ((<< | >>) additiveExpr)*

additiveExpr : multiveExpr ((+ | -) multiveExpr)*

multiveExpr : unaryOprExpr ( (* | / | %) unaryOprExpr)*

unaryOprExpr : *unaryExpr | &unaryExpr | !mathExpr | mathExpr
             | +additiveExpr | -additiveExpr | ++ referenceExpr
             | -- referenceExpr | unaryExpr | (mathExpr)

unaryExpr : (0 ... 9) (0 ... 9) * | True | False | referenceExpr

referenceExpr : varNameExpr | varNameExpr ->. var
              | varNameExpr -> var | varNameExpr [ additiveExpr ]

varNameExpr : var (. var)*

eventNameExpr : var (. var)*
```

Appendix C

Statefule Timed CSP

In the following, we present the syntax and the informal semantics of STCSP processes, based on [143].

A Stateful Timed CSP model (hereafter a model) is a tuple $\mathcal{S} = (Var, Init_G, P)$ where Var is a finite set of finite-domain global variables; $init_G$ is the initial valuation of the variables and P is a timed process. A variable can be of a predefined type like Boolean, integer, array of integers or any user-defined data type. Process P models the control logic of the system using a rich set of process constructs. A process can be defined by the grammar presented in Figure C.1. For simplicity, we assume that P is not parameterized.

Process *Stop* does nothing but idling. Process *Skip* terminates, possibly after idling for some time. Process $e \rightarrow P$ engages in event e first and then behaves as P. Note that e may serve as a synchronization barrier, if combined with parallel composition. In order to seamlessly integrate data operations, we allow sequential programs to be attached with events. Process $a\{program\} \rightarrow P$ performs data operation a (i.e., executing the sequential program whilst generating event a) and then behaves as P. The program may be a simple procedure updating data variables (written in the form of $a\{x := 5;\ y := 3\}$) or a complicated sequential program.

Given a channel ch with pre-defined buffer size, process $ch!exp \rightarrow P$ evaluates the expression exp (with the current valuation of the variables) and puts the value into the tail of the respective buffer and behaves as P. Process $ch?x \rightarrow P$ gets the top element in the respective buffer, assigns it to variable x and then behaves as P. Channels with zero buffer size require synchronous input and output while channels with non-zero buffer size only perform asynchronous input and output operations.

A conditional choice is written as $if(b)\{P\}else\{Q\}$, whereas $ifb(b)\{P\}$ is referred to as a blocking conditional choice that is blocked until b is satisfied and behaves as P. Process $P \mid Q$ offers an (unconditional) choice between P and Q. Process $P;\ Q$ behaves as P until P terminates and then behaves as Q immediately. Process $P \backslash X$ hides occurrences of events in X. Parallel composition of two processes is written as $P \mid\mid Q$, where P and Q may communicate through event synchronization (following CSP rules [74]) or shared

$P = Stop$	–	in-action
$\mid Skip$	–	termination
$\mid e \rightarrow P$	–	event prefixing
$\mid a\{program\} \rightarrow P$	–	data operation prefixing
$\mid ch?exp \rightarrow P$	–	channel input
$\mid ch!x \rightarrow P$	–	channel output
$\mid if(b)\{P\}else\{Q\}$	–	conditional choice
$\mid ifb(b)\{P\}$	–	blocking conditional choice
$\mid P \mid Q$	–	general choice
$\mid P \backslash X$	–	hiding
$\mid P;\ Q$	–	sequential composition
$\mid P \mid\mid Q$	–	parallel composition
$\mid Q$	–	process referencing
$\mid Wait[d]$	–	timed delay*
$\mid P\ timeout[d]\ Q$	–	timeout*
$\mid P\ interrupt[d]\ Q$	–	timed interrupt*
$\mid P\ within[d]$	–	timed responsiveness*
$\mid P\ deadline[d]\ Q$	–	timed deadline*
$\mid e \twoheadrightarrow P$	–	urgent event prefixing *

Figure C.1: STCSP Grammar

variables. Notice that if P and Q do not communicate through event synchronization, then it is written as $P \mid\mid\mid Q$, which reads as "P interleave Q". A process may be given a name, written as $P \widehat{=} Q$, and then referenced through its name. Recursion is allowed by process referencing. Additional process constructs (e.g., while or periodic behaviors) can be defined using the above.

In addition, a number of timed process constructs (marked with * in Figure C.1) are designed to capture common real-time system behavior patterns. Let $d \in R_+$. Process $Wait[d]$ idles for exactly d time units. In process $P\ timeout[d]\ Q$, the first observable event of P shall occur before d time units elapse (since process $P\ timeout[d]\ Q$ is activated). Otherwise, Q takes over control after exactly d time units. In process $P\ interrupt[d]\ Q$,if P terminates before d time units, $P\ interrupt[d]\ Q$ behaves exactly as P. Otherwise, $P\ interrupt[d]\ Q$ behaves as P until d time units and then Q takes over. In contrast to $P\ timeout[d]\ Q$, P may engage in multiple observable events before it is interrupted. Process $P\ within[d]$ must react within d time units, that is, an observable event must be engaged by process P within d time units. Urgent event prefixing [42], written as $e \twoheadrightarrow P$, is defined as $(e \rightarrow P)within[0]$, that is, e must occur as soon as it is enabled. In process $P\ deadline[d]$, P must terminate within d time units, possibly after engaging in multiple observable events. Notice that a timed process construct is always associated with an integer constant d, which is referred to as its parameter.

Appendix D

GLOSSARY

- **Cartesian Semantics** A semantics for concurrent system that is presented by Cartesian vectors [65]. It has been used in the Cartesian partial order reduction algorithm for concurrent programs.

- **CSP (Communicating Sequential Processes)** A formal language for describing patterns of interaction in concurrent systems [74]. It is a member of the family of mathematical theories of concurrency known as process algebras, containing process operators parallel ($\|$), interleave ($|||$), interrupt ($\triangle$), choices ($\Box$ and $\sqcap$), etc.

- **Formal Methods** A particular kind of mathematically based techniques for the specification, development and verification of software and hardware systems. A study of mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems. The phrase 'mathematically rigorous' means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verification are rigorous deductive processes in that logic (i.e., each step follows from a rule of inference and hence can be checked by a mechanical process) [28].

- **Label Transition System (LTS)** An abstract machine consists of a set of states and transitions between states, which may be labeled with labels chosen from a set; the same label may appear on more than one transition. It is adopted to represent the state space of systems for model checking techniques.

- **Linear Temporal Logic (LTL)** Formalism commonly used to describe temporal requirements precisely. There are a few basic operations given with symbols $\Box$, $\Diamond$, X, U, W, R corresponding to English language terms 'always', 'eventually', 'next', 'until', 'weak-until' and 'release'.

- **Liveness Property** Properties defined by LTL formulas, which means that something good eventually happens. Formally, every finite prefix of a counterexample can be extended to an infinite path that satisfies the formula.

- **Model Checking** Given a model of a system, exhaustively and automatically checking whether this model meets a given specification. Typically, one has hardware or software systems in mind, whereas the specification contains safety requirements such as the absence of deadlocks and similar critical states that can cause the system to crash. Model checking is a technique for automatically verifying correctness properties of finite-state systems.

- **NesC (Network Embedded System C)** A component-based, interrupt-driven programming language used to build applications for the TinyOS platform [53]. NesC has been widely used to develop sensor network programs.

- **Partial Order Reduction (POR)** A technique for reducing the size of the state space to be searched by a model checking algorithm. It exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders. In explicit state space exploration, POR usually refers to the specific technique of expanding a representative subset of all enabled transitions. POR has also been described as model checking with representatives [124]. There are various techniques of POR, including the stubborn set method [156], ample set method [124], and persistent set method [59].

- **PAT (Process Analysis Toolkit)** PAT is a self-contained framework for to support composing, simulating and reasoning of concurrent, real-time systems and other possible domains. It comes with user friendly interfaces, featured model editor and animated simulator. Most importantly, PAT implements various model checking techniques catering for different properties such as deadlock-freeness, divergence-freeness, reachability, LTL properties with fairness assumptions, refinement checking and probabilistic model checking. To achieve good performance, advanced optimization techniques are implemented in PAT, e.g. partial order reduction, symmetry reduction, process counter abstraction, parallel model checking. So far, PAT has over 2300 registered users from over 550 organizations in 58 countries and regions.

- **Process Algebra** A diverse family of related approaches for formally modelling concurrent systems. They provides a tool for the high-level description of interactions, communications, and synchronization between a collection of independent agents or processes. They also provide algebraic laws that allow process descriptions to be manipulated and analyzed, and permit formal reasoning about equivalences between processes (e.g., using bisimulation). Leading examples of process calculi include CSP, CCS, ACP, and LOTOS.

- **Safety Property** Properties defined by LTL and also state reachability. Formally, every counterexample has a finite prefix such that, however it is extended to an infinite

path, it is still a counterexample. In our work, we allow safety properties to be defined as LTL formulas, state reachability assertions, etc.

- **Software Verification** A discipline of software engineering with goal to assure that software fully satisfies all the expected requirements. The aim of software verification is to find the errors introduced by an activity, i.e. check if the product of the activity is as correct as it was at the beginning of the activity.

- **Stateful Timed CSP (STCSP)** A formal specification language to model hierarchical realtime systems that extends Timed CSP [43, 134] with language constructs to manipulate data structures and data operations in order to support real-world applications. STCSP supports a rich set of timed process constructs to capture timed system behavior patterns, e.g., delay, deadline, timeout, timed interrupt, etc.

- **TinyOS** A component-based operating system and platform targeting sensor networks. TinyOS is an embedded operating system written in the NesC programming language as a set of cooperating tasks and processes. It is intended to be incorporated into smartdust. TinyOS started as a collaboration between the University of California, Berkeley in co-operation with Intel Research and Crossbow Technology, and has since grown to be an international consortium, the TinyOS Alliance.