N° d'Ordre : 04/2017-D/INF

RÉPUBLIQUE DÉMOCRATIQUE ALGERIENNE

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et Technologie Houari-Boumediene

Faculté d'Électronique et d'Informatique

THÈSE

Présentée pour l'obtention du grade de **DOCTEUR EN SCIENCE**

EN : INFORMATIQUE

Specialté : Informatique Mobile

Par : OUADJAOUT Abdelraouf

Sujet

Fiablité Logicielle dans les Réseaux Embarqués Sans Fil

Soutenue en publique, le 10/05/2017, devant le jury composé de :

Mme.	M. BOUKALA	Professeur à l'U.S.T.H.B.	Présidente
M.	N. BADACHE	Professeur à l'U.S.T.H.B.	Directeur de thèse
Mme.	F. BELALA	Professeur à l'U. Const2	Examinatrice
M.	M. MEZGHICHE	Professeur à l'U.M.B. Boumerdès	Examinateur
M.	M. AISSANI	Maître de Conférence A à l'E.M.P.	Examinateur
M.	Y. HAMMAL	Maître de Conférence A à l'US.T.H.B.	Examinateur
M.	A. MINÉ	Professeur à l'U. Paris6	Invité

Abstract

In this thesis, we endeavor to address the issue of software reliability in wireless embedded networks. In contrast to most existing approaches, we develop *sound* solutions that can *certify* program properties by considering all possible executions without any exception. The foundations of our propositions are based on the formal framework of *abstract interpretation*, a successful theory for the approximation and the analysis of the semantics of complex programs. Using this formalism, we tackle two fundamental and challenging software problems in wireless embedded networks.

The first problem is related to an important *qualitative* issue affecting the safety of these systems. More particularly, we design a sound analysis of *device drivers* for verifying their correct interaction with hardware peripherals. The importance of such a goal derives from the fact that these programs are particularly hard to develop, error-prone and highly critical for the reliability of the system. Our analysis can handle complex functional properties and targets real-world device drivers in TinyOS programs, the *de facto* operating system in wireless embedded networks. We propose an efficient compositional method for analyzing preemptive executions that can scale to complex interrupt-based drivers. In addition, a number of partitioning techniques are also presented in order to track crucial information about the peripheral hardware and the TinyOS tasks queue. The proposed approach has been implemented in a prototype static analyzer and several experiments have been performed on real-world TinyOS device drivers with promising results.

The second problem addressed in this thesis is related to *quantitative* aspects of software reliability. We focus on crucial *performance metrics* of wireless communication protocols (such as energy consumption and packets throughput) and we propose a sound static analysis for inferring guaranteed bounds of these quality indicators. In addition to the soundness of the results, our analysis can provide analytic symbolic bounds expressed as functions of protocol parameters. To do so, we introduce a novel domain of Abstract Markov Chains that provides a finite and symbolic representation to over-approximate the (possibly unbounded) set of stochastic behaviors of the protocol. Our analysis operates in two steps. The first step is a classical abstract interpretation of the source code, using stock numerical abstract domains and a specific automata domain, in order to extract the Abstract Markov Chain of the program. The second step computes symbolic bounds of its stationary distribution using a parametric Fourier-Motzkin elimination algorithm. We present a prototype implementation of the analysis and we discuss some preliminary experiments.

Résumé

Dans cette thèse, nous abordons la problématique de la fiabilité logicielle dans les réseaux embarqués sans fil. Contrairement aux approches existantes, les solutions qu'on propose sont *sûres* par construction et peuvent donc être utilisées pour *certifier* qu'une propriété est verifiée pour toutes les exéctutions possibles. Pour ce faire, nous utilisons la théorie de *l'interprétation abstraite* qui est une théorie très utile pour l'approximation et l'analyse sémantiques de programmes complexes. En se basant sur ce cadre formel, nous abordons deux problèmes fondamentaux et difficiles dans l'analyse des programmes des réseaux embarqués sans fil.

La première analyse est de nature *qualitative* et concerne la sûreté de ces programmes. En particulier, nous nous intéressons à la conception de méthodes automatiques pour la vérification des *pilotes matériels* afin de certifier leur comportement correcte avec leurs périphériques. L'importance d'une telle étude découle du fait que ces programmes sont généralement très critiques, difficile à mettre en œuvre et très susceptibles à contenir des *bugs*. Notre analyse peut traiter des propriétés fonctionnelles complexes sur des pilotes du système TinyOS, qui est le logiciel *de facto* des réseaux embarqués sans fil. On propose une méthode compositionnelle pour l'analyse d'exécutions préemptives qui peut supporter des scénarios d'interruptions complexes. En plus, on présente un ensemble de domaines abstraits par partitionnement qui permettent de préserver des informations cruciales sur le système, telles que l'état du périphérique et le contenu de la queue des tâches de TinyOS. Nous avons implémenté ces techniques dans un prototype et nous avons effectué plusieurs expériences sur des cas réels de pilotes TinyOS avec des résultats très prometteurs.

Le deuxième problème traité durant cette thèse est de nature *quantitative*. Nous nous intéressons à *l'évaluation de performance* des protocoles de communications sans fil dans ces systèmes et nous proposons une analyse statique pouvant inferer des bornes sûres concernant un certain nombre de métriques quantitatives, tels que l'énergie ou le débit. En plus de la propriété de sûreté offerte par l'analyse, les résultats obtenus sont aussi de nature analytique et symbolique, dans le sens où elles sont exprimées en fonctions des paramètres du protocole. Pour ce faire, on introduit un nouveau domaine de chaînes de Markov abstraites qui offre une représentation symbolique et finie servant de sur-approximation d'un ensemble (peut être infini) de comportements stochastiques du protocole. Cette analyse s'effectue en deux étapes. La première est une interprétation abstraite classique du code source en utilisant des domaines numériques standards et un domaine d'automates spécifique, dans le but d'extraire une chaîne de Markov abstraite du programme. Dans la deuxième étape, on calcule les bornes symboliques de la distribution stationnaire de la chaîne en utilisant l'algorithme paramétrique d'élimination de Fourier-Motzkin. Nous présentons un prototype de cette méthode et nous discutons les résultats des expérimentations.

Acknowledgments

This thesis has been a long and extremely enriching journey. Besides the scientific reward of discovering the beauty of the theory of abstract interpretation, I have been pleased to collaborate with great and extremely helpful people. I hope that they will find in these words the expression of my profoundly sincere gratitude.

I would like to thank deeply my adviser Professor Nadjib Badache who offered me the opportunity to discover and practice the subject of wireless embedded networks. Starting from the EPFL internship and until the wonderful WSN lab at CERIST, he has been always supporting my efforts while giving me enough autonomy to progress. I had also the wonderful and priceless opportunity to work with Professor Antoine Miné who enlightened my understanding of the theory of abstract interpretation. With his precious and unique experience in this field, he has always helped me to overcome challenges in my thesis and make the right decisions. I consider myself a very lucky person for being able to work with him and I would like to thank him very deeply. My Sincere thanks go also to the thesis jury Pr. Malika Ioualalene-Boukala, Pr. Mohamed Mezghiche, Pr. Faiza Belala, Dr. Mohamed Aissani and Dr. Youcef Hammal for accepting to review this work and for their valuable comments and feedback.

I would like to thank my great friends and workmates at CERIST for their continuous support, valuable discussions and sincere friendship: Noureddine Lasla, Mohamed Amine Kafi, Messaoud Doudou and Mildou Bagaa. During my thesis, I also spent a pleasant year within the APR team at LIP6 and I would like to thank the funny "doctorants" team and all the other members: Vincent Botbol, Aurélien Deharbe, Matthieu Dien, Rémy El-Sibaie, Matthieu Journault, Frédéric Peschanski, Marie Pellau, Thibault Suzanne, Steven Varoumas, Véronique Varenne and Ghiles Ziat.

My parents are a very precious treasure in my life and I would like to dedicate this thesis to them as an expression of my gratitude for everything they did for me. Also, I wish to thank my dear brothers Zaki, Houssem and Aymen. Last but not least, I would like to thank my wife Afaf who was always with me during this journey providing its comforting and precious support. *Thank you Afaf for your love and patience ...*

Contents

1 Introduction **1**

1.1 Context 1

1.2 Motivation 2

1.3 Our Contributions 3

1.4 Organization 5

2 State of the Art **7**

2.1 Wireless Embedded Networks 7

2.2 Software Architecture 8

2.2.1 Language 10

2.2.2 Execution 10

2.3 Software Safety 12

2.3.1 Language Runtime Errors 13

2.3.2 Functional Properties 13

2.3.3 Examples 14

2.4 Software Metrics 15

2.4.1 Energy 15

2.4.2 Throughput 16

2.4.3 Memory 16

2.5 Verification Approaches 16

2.5.1 Testing 16

2.5.2 Model Checking 18

2.5.3 Deductive Methods 20
2.5.4 Abstract Interpretation 21
2.6 Discussion 22
2.7 Conclusion 23

3 Abstract Interpretation **25**
3.1 Preliminaries 25
3.1.1 Lattices 26
3.1.2 Fixpoints 26
3.1.3 Galois Connections 27
3.1.4 Fixpoints Revisited 30
3.2 Abstract Interpretation Tutorial 31
3.2.1 Language 31
3.2.2 Concrete Semantics 32
3.2.3 Abstract Semantics 34
3.2.4 Soundness Proofs 37
3.3 Conclusion 39

4 Safety Verification of Device Drivers **41**
4.1 Introduction 41
4.2 TinyOS Device Drivers 43
4.3 Assumptions and Notations 47
4.4 Sequential Executions Analysis 48
4.4.1 Concrete Semantics 48
4.4.2 Abstract Semantics 50
4.5 Preemptive Executions Analysis 58
4.5.1 Concrete Semantics 58
4.5.2 Abstract Semantics 61
4.6 Experiments 66
4.6.1 Test Cases 67

4.6.2 TinyOS Benchmarks . . . 69
4.6.3 Comparison with i-CBMC . . . 72
4.7 Conclusion . . . 73

5 Quantitative Verification of Communication Protocols 75
5.1 Introduction . . . 75
5.2 Concrete Semantics . . . 79
5.2.1 Markovian Traces . . . 80
5.2.2 Semantics Domain . . . 81
5.2.3 Stationary Distribution . . . 82
5.3 Abstract Semantics . . . 83
5.3.1 Abstract Automata . . . 84
5.3.2 Abstract Scenarios . . . 88
5.3.3 Abstract Markov Chains . . . 90
5.4 Proportional Expectations . . . 93
5.5 Resolution . . . 96
5.6 Experiments . . . 98
5.7 Conclusion . . . 100

6 Conclusion & Perspectives 101
6.1 Conclusion . . . 101
6.2 Perspectives . . . 102

Chapter 1

Introduction

The technology of embedded systems has become an integral part of our daily life activities. Those systems can range from small computing nodes that periodically monitor the environment climate, to high-end embedded computers that can perform complex *in situ* processing for controlling critical industrial infrastructures. In recent years, the novel paradigm of *Internet of Things* (IoT) has changed these technologies from isolated and closed components into a world of interconnected smart devices. IoT opens many exciting directions for novel and rich applications where everyday *things* will be able to connect to the Internet in order to interact with their context or with other objects for a better decision making process.

1.1 Context

In this work, we are interested in a core component of IoT which is the technology of *wireless embedded networks*. A wireless embedded network is composed of a set of independent computing entities (*wireless nodes*) with limited resources that can communicate with each other wirelessly. These nodes are able to auto-organize themselves to build an *ad hoc* network that can cover large geographic areas for collecting specific measurements or controlling remote equipments. The operation of the network is maintained by a distributed program that is embedded into every wireless node. However, these programs are generally error-prone and software failures are a common and dangerous symptom in such systems. This is due to their complex software stack that combines the intricacies of low-level hardware programming and the difficult management of the distributed and lossy networks.

Consequently, it is vital to ensure the reliability of these programs through automatic analysis tools that help developers discover the anomalies of their implementa-

tions. Software reliability can be addressed from two distinct perspectives. Firstly, a reliable program should verify the functional specifications of the runtime environment and the safety rules for proper hardware operations. These *qualitative* requirements of software reliability prevent the emergence of undesirable software bugs during execution. Secondly, a reliable program is also required to exhibit acceptable performances that preserve the resources of the system, in terms of memory, energy and bandwidth. A program that does not consider such *quantitative* aspects can exhaust the available resources which may disconnect the system.

1.2 Motivation

The research community has been very active during the last decade targeting these two fundamental problems and many solutions have been proposed. Nevertheless, less attention has been given to *sound* approaches that can cover all possible executions and provide formal guarantees about the obtained analysis results. To better explain this important point, let us address the qualitative and quantitative parts separately and discuss succinctly their related work.

Existing safety analysis for wireless embedded programs are either unsound[1] [Reg05, CAE+07, LR10, SLA+10, MVO+10, BK11, Buc12, KLM+15] or limited to language runtime errors [CR06, BNS10] such as out-of-boundaries array access or null pointer dereference. To our knowledge, no existing sound verification tool is able to hunt *hardware interaction errors* that occur when a device driver fails at handling correctly the management of the hardware. These errors are however recurrent in computing systems in general (for example, 85% of Windows XP bugs were caused by incorrect device drivers [OT03]), and embedded systems are not an exception. More importantly, the presence of such errors can have dramatic consequences on the system, since a wireless node is continuously communicating with its peripherals and therefore can not operate without correct hardware interaction mechanisms.

Quantitative aspects in program analysis are a more recent research topic, specially when considering soundness in mind [MMHS11, SCG13, CS13, CS14, BEFFH16, BGP+16]. However, the paradigm of wireless embedded networks has its own particularities that can not be handled by the existing sound approaches. Indeed, these approaches focus on quantifying the probability of reaching error states or the expectation of some program variables. In wireless embedded programs, we are also interested by computing another type of information, which reflects the change of some metric *proportionally* to one time unit. These velocity metrics are extremely important

[1] In this work, we consider that an approach is sound if it can cover all executions within finite time.

for system designers and include for example the throughput (the number of packets sent in one second) and the duty cycle (proportion of time the wireless transceiver is activated). It is worth noting that some existing quantitative tools such as PRISM [KNP11] can compute this form of expectation, but are limited to finite state systems and therefore may not be sound when analyzing programs with complex control flows and data structures.

1.3 Our Contributions

In this thesis, we tackle these two fundamental problems and we propose sound verification approaches for (i) ensuring safety functional properties of device drivers and (ii) computing guaranteed bounds of proportional performance metrics in the context of embedded wireless systems. Our analysis is developed within the framework of abstract interpretation, a theory that has been deemed successful for analyzing the semantics of complex and large industrial programs. The main advantages of this theory are its full automation and complete coverage of all possible executions. However, due to the undecidability of program verification problem, it is possible that the analysis considers also spurious executions, which leads to eventual false alarms (for the safety part) or coarse metrics approximations (for the quantitative part). That is said, the theory offers interesting tuning mechanisms that help adjusting the precision of the analysis and reducing the effects of these over-approximations.

In the following, we list the contributions related to each part of our thesis:

Safety. Our device driver analysis is tailored to programs running under the TinyOS operating system, considered as the *de facto* software platform for wireless embedded networks. The analysis makes the following contributions.

- Firstly, it can handle complex functional properties specifying how the driver should interact with the hardware correctly, which is done through a powerful automata-based formalism.
- Secondly, our approach supports the analysis of concurrent executions through arbitrary interrupt preemption. The analysis is performed in a modular and compositional way that analyzes every interrupt independently and aggregates their results to over-approximate the effect of preemption. By doing so, we avoid reanalyzing interrupts in every context where they are enabled which improves considerably the scalability of the solution.
- Thirdly, we propose a number of partitioning techniques specially tailored to

TinyOS device drivers. They provide a variety of tradeoffs between precision and efficiency and allow keeping crucial information about the internal scheduling policy of TinyOS, the hardware state and interrupt masks.

- Finally, these mechanisms have been implemented in a prototype analyzer that has been able to verify complex and real-world device drivers from the official TinyOS distribution with promising performances in terms of speed, memory consumption and precision.

Performances. We also propose a novel sound static analysis for obtaining bounds of several important proportional metrics, such the energy consumption and the throughput. In summary, the contributions of our approach are:

- The returned bounds are *sound* so they cover every possible executions, and *parametric* in the sense that we obtain symbolic expressions as functions of the protocol parameters.

- We introduce a novel notion of *Abstract Markov Chains* that can over-approximate a set of (possibly unbounded) stochastic behaviors. These abstract chains can capture many essential semantic aspects of communication protocols, are inferred automatically by analyzing the source code of the program and are guaranteed to have a finite size.

- Also, we present a *soundness result* that allows us to derive from the abstract Markov chain a system of parametric linear inequalities for bounding the probability distribution of the steady state behaviors of the protocol. As we will describe later, this stationary distribution is the key element to compute proportional performance metrics.

- Finally, we also present a *method for solving systems of parametric linear inequalities* using a symbolic version of the standard Fourier-Motzkin elimination algorithm. This method allows us to derive the symbolic bounds of the metric of interest.

We should notice that this analysis is still a *work in progress* and can not at the moment verify real world implementations. Nevertheless, it has been tested on a simple C-like language and we believe that the proposed theoretical contributions and the preliminary experimental results open promising research directions for future work.

1.4 Organization

The second chapter of this thesis provides a state of the art of the developed methods for verifying programs in wireless embedded networks. We explain the general software architecture of these systems and we show how it can be affected by different forms of programming errors. We highlight the fundamental ideas of the existing solutions and we compare their strengths and weaknesses. The third chapter is devoted to the theory of abstract interpretation, since it represents the foundation of our work. We present briefly the mathematical background of the theory and we explain how it can be applied to the analysis of programs through a didactic tutorial exemplifying the build process of a typical abstract interpreter. The fourth chapter describes in details our static analysis of device driver in TinyOS programs. We present a number of abstract domains that provide different levels of precision. We demonstrate the effectiveness of our approach through the discussion of several experimental results conducted on real-world device drivers. The quantitative analysis is explained in the fifth chapter. We show how the program semantics can be approximated with an abstract Markov chain and how we can extract from it guaranteed bounds of performance metrics. Finally, this thesis is ended by a conclusion summarizing the presented ideas and presenting some future work.

Chapter 2

State of the Art

Ensuring the correctness of programs has represented a long-standing challenge for computing science that remains unsolved until this day. In fact, Turing has proven at a very early stage, in 1949, that program verification is an undecidable problem. This means that there exists no algorithm that can verify in a finite time any given property of a program in general. Consequently, the proposed solutions have been oriented to focus on particular families of programs and targeting specific forms of software failures, in order to alleviate the intractability of the problem.

In this chapter, we provide a comprehensive review of the verification techniques targeting wireless embedded networks. We begin by presenting an overview on wireless embedded networks and we focus on their software architecture by describing the essential building elements composing a typical program controlling such systems. Afterwards, we discuss the different software issues that can disturb their proper execution and also the set of software metrics for quantitatively comparing different implementations. Finally, we highlight the fundamental ideas of the existing approaches for verifying these programs and ensuring the software reliability of the system.

2.1 Wireless Embedded Networks

A wireless embedded network is a distributed system within which small embedded nodes cooperate in order to sense some aspects of their physical environment and propagate their measurements to end users. Additionally, these systems can allow a remote control of wireless nodes to perform some physical action depending on the target application.

To illustrate how these systems are used, we consider the case of a *precise agriculture*

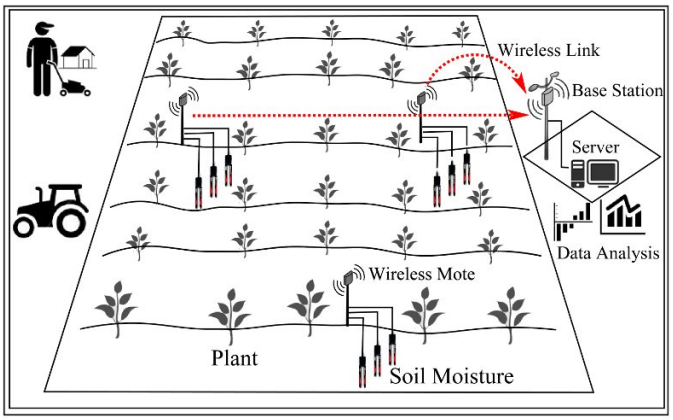


Figure 2.1: An example of a wireless embedded network for precise agriculture.

system that aims at reducing water waste used for irrigation. Indeed, the use of traditional irrigation techniques considerably impacts water preservation: more than %50 of water would be lost in the irrigation use [FAO03]. Moreover, the use of traditional irrigation techniques has a negative effect on the environment, such as the salinization, intrusion of brackish water, *etc.* For this purpose, efficient irrigation control systems based on a wireless embedded network have been proposed to reduce theses negative impacts on the environment [PRP+06, WHWX07, PE08]. An illustrative architecture is depicted in Figure 2.1 where wireless nodes are deployed in the crop field equipped with a set of soil moisture sensors. The sensor nodes detect the amount of water in the soil via the soil moisture sensors. To give an accurate detection, the soil moisture sensors may be placed at different levels of the ground. Periodically, the sensor nodes measure the amount of water in the soil and then communicate the measured values to a control station (*i.e.*, the base station). The collected report messages can be used for automatic control of solenoid irrigation valves, or saved in a database for post-analysis.

The portfolio of this paradigm includes several other applications, such as air quality monitoring [HVL+10], parking systems [PPH12], *etc.* More recently, more home-centric and consumer-oriented applications have emerged in the ecosystem of the Internet of Things. In such applications, humans, using their smartphones, interact with their daily devices in order to get instant information (such as the location of lost keys using a BLE proximity beacon) or to perform a remote control (such the modification the ambient temperature using a smart thermostat).

2.2 Software Architecture

In general, a wireless node has very constrained hardware capabilities that limit its performance, as illustrated in Figure 2.2(a). In most cases, a node is chipped with a low-

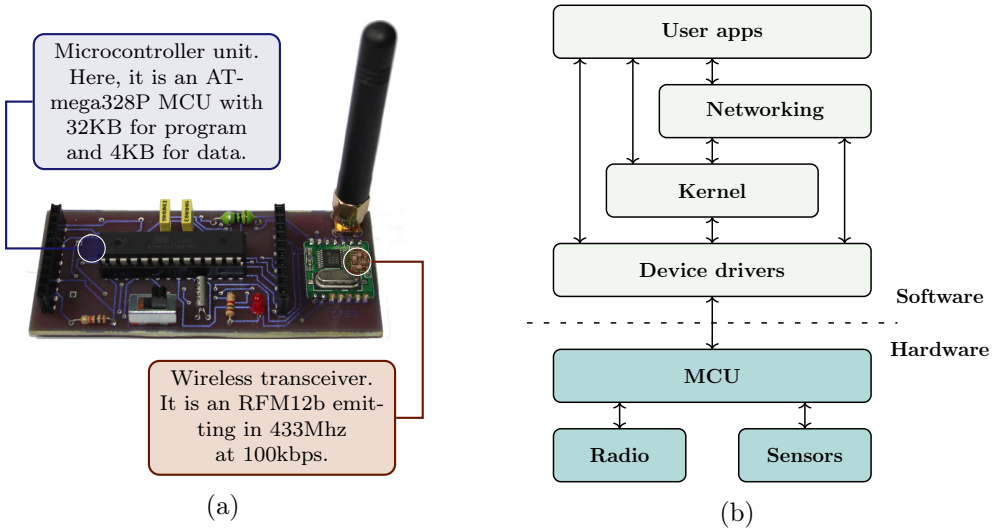


Figure 2.2: Architecture of a typical wireless embedded node. (a) Hardware architecture. (b) Software architecture.

power microcontroller unit (MCU) containing only a few kbytes of memory and running at slow frequencies, as low as 16Mhz. In addition, it embeds a wireless transceiver that can not send more than few hundreds of kilo-bits per second. The reason behind these *downgraded systems* is to reduce their energy consumption, which allows their deployment for long periods with limited energy supplies. In addition, production costs are narrowed which is a crucial objective when covering large areas that require a significant number of nodes to be deployed.

In such constrained systems, software should be designed carefully. Every layer in the software stack must be optimized to run correctly under the limitations of memory, speed and energy. Consequently, developing a custom software stack *from scratch* each time designing a new system is painful and error-prone. For this reason, many tiny operating systems have been developed that facilitate the development in such constrained environments. Figure 2.2(b) depicts the typical architecture of an embedded application and how it fits within the software stack of the operating system. Basically, the OS provides a minimalist kernel that orchestrates the execution of the different components of the software stack. In addition, it offers a set of libraries, such as device drivers and networking protocols, which facilitates the rapid development of applications that necessitate complex features. To do so, the OS architecture should be designed in a modular way by specifying an abstract API that software components should implement. By doing so, the modules of the software stack are decoupled and the overall architecture can evolve easily without requiring a great amount of code modification.

Before presenting the different software failures that can arise during program exe-

cution, it is important to understand the internals of the OS, because the execution flow of the program is controlled by the operating system. We limit the description herein to the most popular system: TinyOS, an event-based operating system developed by Levis et al. [LMP+04] for low-power wireless sensor nodes. It supports a variety of hardware platforms with built-in device drivers, networking protocols, security mechanisms, *etc.* TinyOS benefits from a large research community that contributes continuously with many third-party libraries that extend the scope of the OS to various application fields.

2.2.1 Language

TinyOS programs are written in the nesC language [GLvB+03], a dialect of C that offers a modular programming paradigm for flexible organization of software components. Modularity of nesC relies on the concept of *static virtualization* that can be explained as follows. A nesC program is composed of a set of *modules* that are decoupled from each other using a form of parametric polymorphism. In other words, when a module requires a certain external functionality provided by another module, it does not call it directly but makes use of an abstract *interface*. The implementation of the interface can be provided by several other modules, and the developer can *wire* between the calling module and the actual implementation via a *configuration* file. Such decoupling provides great flexibility to the software stack since different implementations for the same abstract service can be interchanged easily.

To explain further these notions, let us consider the example depicted in Figure 2.3. In the declaration of the module `RoutingP`, the interface `Send` is decorated with the `uses` keyword which indicates that this module can call the command `Send.send` and requires an implementation of it. Several modules can provide this functionality and may differ in their implementation. For example, the module `AsyncMacP` implements an asynchronous MAC protocol that is suitable for event-based applications, whereas the module `SyncMacP` provides a synchronous communication that fits better for periodic applications. The developer can choose among the available implementations by defining the appropriate wiring in the configuration file `RoutingC`. It is worth noting that, at compile-time, this abstract wiring links between modules are statically transformed into ordinary function calls which represents an efficient implementation of polymorphism.

2.2.2 Execution

During execution, TinyOS programs are driven by a two-level preemption based on the concepts of interrupts and tasks. *Interrupts* represent the high priority preemption

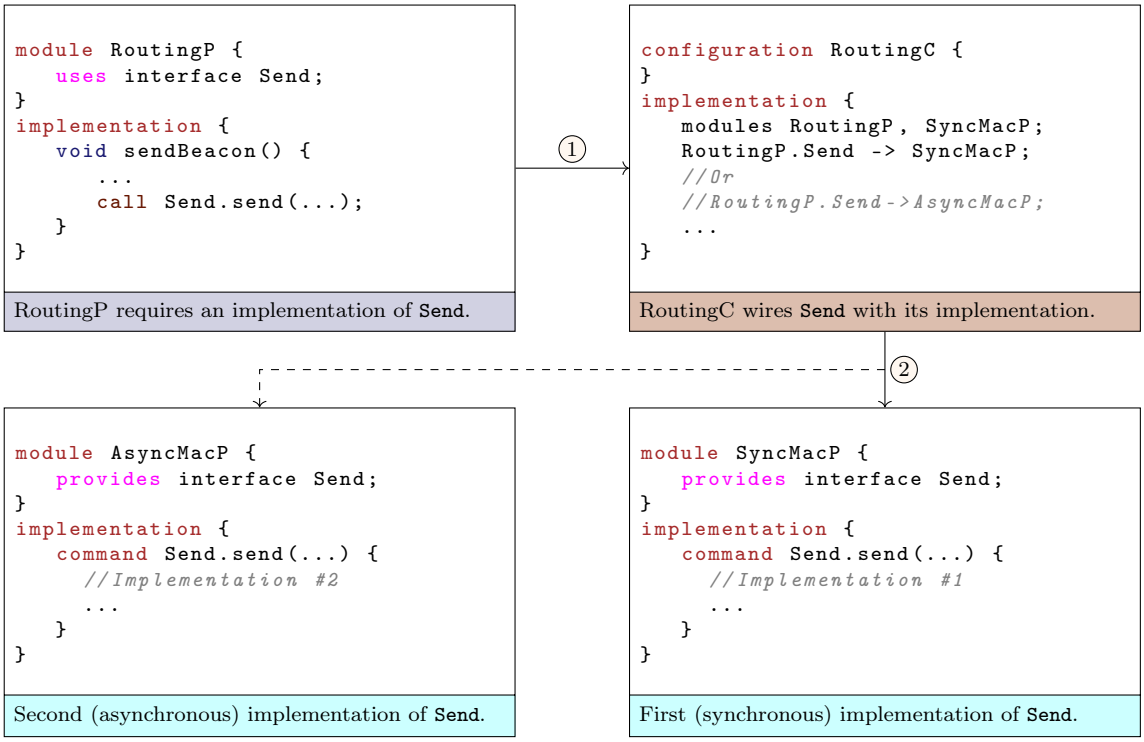


Figure 2.3: Example of static virtualization in TinyOS programs.

level. They play an important role in designing power-efficient programs and are used to free up the MCU from actively waiting for the occurrence of a particular event. During these waiting periods, the microcontroller can either enter various sleep modes to save energy or execute other *waiting functions* to save time. *Tasks* are a special feature of nesC that provides this concept of *waiting functions*. This mechanism allows postponing the execution of a function in order to let other tasks execute. That is, when a task is *posted*, the TinyOS scheduler puts it in a *task queue* and the execution of the current function is resumed. The scheduler, at specific moments, checks its task queue in order to *consume* the posted tasks. Tasks run at low priority and can not preempt each other, while interrupts can preempt the execution of tasks or other interrupts.

To explain further this execution model, we show in Figure 2.4 the different steps of a TinyOS program lifecycle. These steps can be divided into two main phases: the initialization phase and the infinite loop. The initialization phase is responsible for bootstrapping the different software components. At the beginning, the kernel and the device drivers are initialized. These steps are executed without enabling interrupts, allowing the system to start in a more controlled manner. After that, the TinyOS kernel consumes the tasks that have been posted by device drivers and terminates the initialization phase by starting the user applications. This final step is executed with

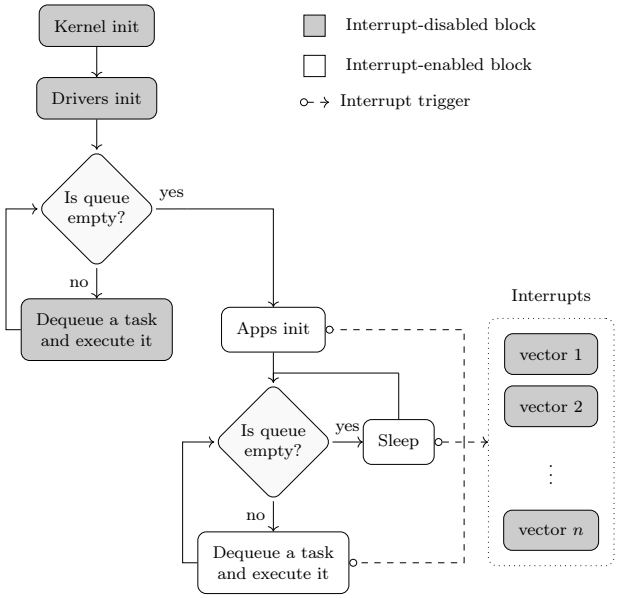


Figure 2.4: The execution model of a TinyOS program.

interrupts being enabled because some drivers rely on interrupts for proper operation.

The second phase is the infinite loop that constitutes the most important proportion of the program's lifetime. This phase begins by consuming the previously posted tasks. When the tasks queue becomes empty, the MCU can enter the sleep mode in order to save energy while waiting for interrupts. After the occurrence of an interrupt, the MCU executes its corresponding handler function. The particular sequence *tasks-sleep-interrupts* forms the body of the infinite loop which is repeated indefinitely until the shutdown of the system. It is important to note that interrupts do not occur only during sleep periods, but they can preempt the execution of the program in every control location when specific conditions on some hardware registers are met.

2.3 Software Safety

The development of a safe and correct embedded program is a laborious task. Software failures are indeed a common symptom in these systems, due to the architectural complexity of the software stack, the burdening low-level details of embedded programming and the error-prone human nature. Numerous deployments of wireless sensor networks have demonstrated that intriguing and complex bugs may appear in long term runs [BISV08, LBV06, WALJ+06]. Although careful inspections were performed prior deployment, these systems suddenly crashed (after just few days in some cases) since they operated in new and unexpected settings different from the lab context.

In this section, we elaborate a taxonomy of potential bugs in a wireless embedded program. We discuss some of their impact by exemplifying some real bugs in the TinyOS code base and providing technical insights about their origins, which is extremely helpful for building special-purpose verification tools targeting effectively specific error types.

2.3.1 Language Runtime Errors

One of the main tasks of a compiler is to detect errors that violate the specification of the programming language. The verification performed by compilers is done at three levels: lexical, syntax and static semantics. However, the specifications of programming languages also describe another error level, called the *runtime errors* (RTE), that are not static but emerge in some particular execution contexts. For the case of the C language, there exists many runtime errors that can have dramatic consequences. An important and commonly recurrent example is the out-of-bound array access that occurs when the index expression of an array access evaluates to a value greater than the actual array size. Another famous example is the misuse of the modular arithmetics of integer variables that occur when developers do not consider the limited-size of variables by assigning to them too big values.

2.3.2 Functional Properties

A software component that does not violate the language specifications is not necessarily correct. Indeed, a component encapsulating a given functionality can fail in delivering the correct result for every possible input, albeit all used statements are correct with respect to the specifications of the programming language. These errors arise from violation of the functional specification of the desired logic, due to incorrect flow of computations or from an erroneous usage of the system APIs. Two particular forms of such errors have been addressed separately due to their importance:

Hardware Interaction Errors. An embedded program continuously interacts with the peripheral hardware, such as the embedded sensors and the wireless transceiver. These interactions should obey a set of rigorous rules that specify how the hardware functionalities can be accessed by a program. Such specifications are generally described in an informal form by manufacturers in datasheets that can contain a large amount of low-level details spread over hundreds of pages. Consequently, device drivers, which primary goal is to effectively interact with the hardware, may fail to encapsulate the appropriate interaction pattern as specified in datasheets. Faulty device drivers are an

important source of failures and can cause dramatic situations, such as data corruption or system crash.

Communication Protocol Errors. Effective cooperation among the network nodes is a cornerstone element in the operation of a wireless embedded network. Nodes interact via message exchange protocols that establish a distributed behavior aiming at achieving a certain task through nodes collaboration. Some of these interactions can be local at the direct neighboring of a node, such as MAC protocols that ensure correct access to the wireless medium. On the other hand, network-wide protocols require to establish multi-hop communications with distant nodes. Failing at implementing correctly a network protocol can have several negative impacts on the system, such as node disconnection and energy exhaustion.

2.3.3 Examples

Let us exemplify some of these errors by real bugs found and fixed in the TinyOS codebase. The first example shows the case of a language runtime error that emerges from an unexpected modular arithmetic in an unsigned 8-bit variable that receives a value larger than `0xff`.

Example 2.1. The following code fragment is from the CC1000 driver in TinyOS 2.x:

```
1  void rxData(uint8_t in) {
2    cc1000_header_t *rxHeader = getHeader(rxBufPtr);
3    uint8_t rxLength = rxHeader->length + offsetof(message_t, data);
4    // Reject invalid length packets
5    if (rxLength > TOSH_DATA_LENGTH) {
6      ...
7      return;
8    }
9    ...
10 }
```

The error is located at line 3 and occurs when the transceiver receives a corrupted packet with size `0xff`. The addition will wrap around and the variable `rxLength` will contain an erroneous value. The consequence of this bug is that corrupted packets are not filtered properly at line 5 which may harm the communication stack. ○

This example shows also that software failures can have different semantics and impacts, since this language error can evolve into a communication error because the networking protocols may be affected by this inappropriate filtering of malformed packets. The following example illustrates a case of a hardware interaction error.

Example 2.2. Let us consider the following code fragment from the timer driver of the ATmega128 MCU in TinyOS:

```
1 void setOcr0(uint8_t n) {
2    while (call TimerAsync.compareBusy());
3    if (n == TCNT0)
4       n++;
5    ...
6    OCR0 = n;
7 }
```

This function updates the register `OCR0` (Output Compare Register 0) to a new value in order to prepare the next timer interrupt. In the datasheet of the ATmega128, it is stipulated that "*if the new value written to OCRx is lower than the current value of TCNTx, the counter will miss the compare match*". In some particular situations, when the node experiences a very heavy load as reported in [Ox12], the value returned by the counter register `TCNT0` at line 5 may be greater than `n`. Consequently, the check of equality is not sufficient resulting in the violation of the previous rule and in the complete destruction of the timer synchronization. ○

2.4 Software Metrics

A safe software is not necessarily a *reliable* software. Beside the language and functional properties described previously, there exist several non-functional software metrics employed by system designers to *quantify* the quality of a program. In the context of wireless embedded networks, three major performance indicators are frequently used, and are all related to the way the program manages its available resources.

2.4.1 Energy

Energy is a scarce resource in wireless embedded networks since nodes operate generally on limited energy supplies (batteries, solar panels, etc.). Every system operation consumes some amount of energy, but it is well-known that the wireless transceiver is the most energy demanding. This is the reason why for measuring energy consumption we generally focus on the protocol *duty cycling*, which corresponds to the proportion of time the wireless transceiver is activated. Nevertheless, computing such indicator is not always an easy task, since protocols implement generally complex and dynamic synchronization mechanisms in order to organize which nodes can access the shared links for avoiding collisions.

2.4.2 Throughput

The throughput quantifies the ability of the program to transmit data over the wireless medium, which represents a shared resource among neighboring nodes. It can have several definitions depending on the considered level of communication. For example, the transmission throughput represents the average quantity of information that system can send during one time unit. By considering link drops due to wireless link quality or congestion in multi-hop communications, we can also define a delivery throughput that measures how many packets are effectively received by the end point.

2.4.3 Memory

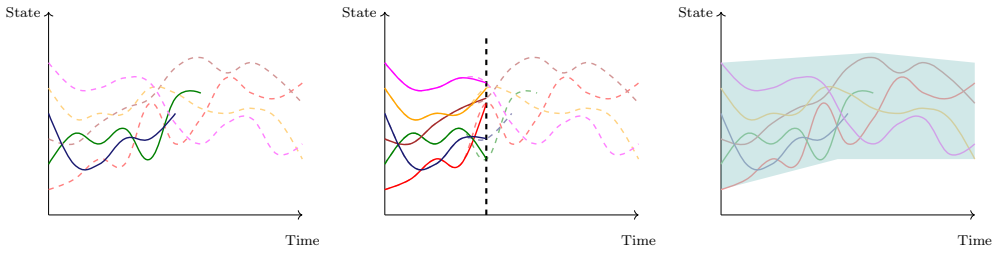
Another important resource to consider is memory, limited to a few hundreds of kylobytes in most systems. As previous metrics, memory consumption is not static and changes continuously during execution. It is therefore vital to ensure that at any moment the application will not require more memory than available. However, in most systems, this change affects only the execution stack (which is related to memory allocated at function calls), since `malloc`-like functions are not available generally.

2.5 Verification Approaches

Due to the undecidability nature of the verification problem, no generic algorithm exists for finding *a priori* every possible error in a program. In fact, the verification process is hampered by the problem of *state space explosion*, meaning that the number of executions that the analyzer needs to go through can be extremely large, at an order that a computer can not handle it. Consequently, effort of the research community has been focused on developing approximation techniques that provide partial answers about specific properties related to program behavior. In this section, we provide an overview of the most important techniques developed so far, and we explain how they were adapted for the context of wireless embedded networks.

2.5.1 Testing

The most widespread program verification technique is *testing* that consists in the assessment of a limited number of finite program execution traces in order to look for the presence of some predefined errors. Implementing a testing technique is generally simpler than other formal methods since it relies on an algorithmic observation of a finite set of executions. However, the results are not *sound* and provided without any



(a) Testing (b) Model checking (c) Abstract interpretation

Figure 2.5: Illustration of verification approaches.

guarantee, since the search space is not entirely covered.

This concept is depicted in Figure 2.5(a). We simplify the representation of program executions with a two dimensions plot that is just a hypothetical view on the evolution of the program state over time. The x-axis of the plot is the execution time flow and the y-axis is a numerical hashcode encoding the program state. When using a testing approach, we observe that we analyze only a finite set of finite length executions, whilst the remaining part of the behaviors space (shown as dotted lines) is simply ignored.

Several techniques have been developed to test a program. The most employed one in the context of wireless embedded networks is *runtime verification* that consists in *instrumenting* the program by modifying (manually or automatically) its source code, in order to add specific control statements that implement some runtime detection mechanism. The aim of this runtime is to monitor the program state at specific control locations, put the program into a safe mode whenever an error is detected and notify the user about the situation. Instrumentation is therefore a transformation technique to augment the program with a self-detection mechanism of error situations.

Safe TinyOS. [CAE+07] is an example of an instrumentation tool for TinyOS to ensure memory safety at runtime. Safe TinyOS is based on Deputy [CHA+07] for inserting appropriate checks at susceptible memory accesses. By providing developers with an annotation system, program statements can be decorated with additional type information for guiding Safe TinyOS inferring the required checks of memory access.

Example 2.3. For example, an array access `x = a[i];` is transformed into:

```
if (i >= len)
  deputy_fail();
x = a[i];
```

where `len` is the size of the array `a`, that is computed statically from the declaration of the variable, or provided by the developer with an annotation. When one of these

assertions is violated during execution, the current program location is identified and notified to the user by three means: mote LEDs, radio and serial messages. ○

Temporal monitors. Bucur [Buc12] proposed another testing approach that allows the runtime verification of LTL (Linear Temporal Logic) properties expressed over program variables and particular control locations (such as function calls, tasks consumption, etc.). To assess the validity of the LTL property at runtime, it is first transformed into a nesC module, called a temporal monitor, that implements an equivalent Büchi automaton accepting the same words (traces) as the LTL formula. The alphabet of the associated language is composed of the set of system events that may affect the status of the formula. To track the occurrence of these events, manual modifications of the program is required.

RID. When performing testing in an emulation environment, a fundamental problem arises: how to schedule interrupts in a consistent way? This question has been addressed by the tool RID [Reg05] that proposes a random testing technique using a *restricted interrupt discipline*, which guides the scheduling of interrupts in order to fire them at semantically valid moments and avoid spurious executions. This is performed by instrumenting the program with explicit calls to interrupt vectors at specific control locations, which helps focusing the analysis on the detection of runtime errors caused by unexpected interrupts.

PM. The previously described testing solutions are only applicable to safety properties. Despaux [DSL14, FDSL15] presented a quantitative testing technique for estimating the end-to-end delay (and therefore the throughput) of communication protocols in Contiki [DGV04], which is another famous operating system for wireless embedded networks. The idea is to instrument the program in order to generate an event log that traces the observed state transitions of the protocol at runtime. By using PM (Process Mining) techniques, a discrete Markov chain is inferred from the event log and the expected end-to-end delay is computed using standard quantitative properties of Markov chains.

2.5.2 Model Checking

Model checking [CE82, QS82] is a family of algorithmic methods for the systematic and exhaustive analysis of finite states systems. The numerous techniques developed by the model checking community allow verifying complex forms of properties, with rich and various logic (temporal, probabilistic, etc.). In some particular cases, model checking

provides more confident and trustworthy results than testing since it can exhaustively search the state space for finite models. However, in general, this technique can not cope completely with the state explosion problem and can not scale to any program size.

Let us return to our previous illustration of program executions in Figure 2.5. The set of executions covered by a bounded model checker is depicted in Figure 2.5(b). We clearly observe that all prefixes to only a certain length are considered. Therefore, not only infinite traces are ignored, but also finite ones with lengths greater than the prefix bound parameter.

Nevertheless, even with a finite search depth, the number of paths to explore can be extremely large when considering interrupt preemption and/or network interactions. To alleviate this problem, *partial order reduction* [CGMP99] is generally employed which is an optimization method that is useful for the analysis of concurrent systems. The technique is based on detecting concurrent behaviors that produce the same result whatever is the order of execution. Therefore, we avoid analyzing all possible interleavings, so we reduce the impact of state explosion.

Anquiro. Mottola et al. proposed Anquiro [MVO+10] as a domain-specific extension to the Bogor model checker [RDH03] that adds support for the specificities of Contiki programs, such as timers and wireless message processing. Basically, the main idea of Anquiro is to translate the semantics of the Coniki program into a finite state machine that can be processed by the Bogor model checker. The user can express a LTL formulae specifying a property about the program's variables and that can be quantified over the network nodes to model correctness rules of communication protocols.

KleeNet. Another model checker for Contiki is KleeNet [SLA+10]. It is based on the symbolic virtual machine KLEE [CDE08] that can execute a program on symbolic inputs instead of randomly selected ones. The symbolic values are propagated along the execution traces and modified by assignments and tests. At the end, we can obtain a path condition expressed in terms of symbolic inputs that leads to some specified bug location. KleeNet extends this symbolic infrastructure to a network setting in order to find node interaction bugs, by injecting specific failures (such as packet loss and node crash) in a nondeterministic way during the symbolic execution of the program.

T-Check. Li and Regehr proposed T-Check [LR10], a bounded model checker of TinyOS programs for verifying safety and liveness properties of communication protocols. The particularity of T-Check is that it uses random walks at specific stages of the verification process in order to alleviate the problem of state space explosion. In

addition, it embeds a partial order reduction technique to avoid exploring redundant paths.

TOS2Cprover. Bucur and Kwiatkowska proposed TOS2CProver [BK11] which transforms TinyOS programs into C programs that can be handled by CBMC [CKL04], a well-known bounded-model checker for C. The tool is tailored to the MSP430 microcontrollers platform. Before applying CBMC, it instruments the source with appropriate assertions expressing generic language safety rules or particular requirements on hardware registers values. Since CBMC can handle only sequential programs, TOS2CProver applies a sequentialization technique by inserting nondeterministic calls to interrupt handlers at specific locations. The interrupt locations are inferred using a partial order reduction in order to decrease the program's state space.

i-CBMC. A different model checking approach has been proposed by Kroening et al. [KLM+15] to overcome some limitations of the partial reduction technique. Indeed, their tool i-CBMC supports native modeling of interrupt preemption using a partial order encoding instead of a partial order reduction. The basic idea of i-CBMC is to enrich the trace formula of CBMC with an encoding of the possible interrupts interleavings using a set of symbolic clocks. These clocks represent the logical occurrence time of the access events on the program variables. The proposed method defines a set of constraints for restricting the values of theses clocks and expressing a set of happens-before conditions emerging from the semantics of interrupt preemption. Using these constraints, a partial order among interleavings can be defined and the search space is therefore reduced.

PRISM. In the category of quantitative analysis, PRISM model checker [KNP11] is considered as one of the most successful tools and has been employed to analyze many sensor network protocols such as IEEE 802.15.4 [Fru06] and the gossiping routing protocol [GMR+13]. It supports several stochastic models (discrete and continous time markov chains, markovian decision processes and probabilistic timed automata), but is limited to finite state systems. PARAM [HHZ11] is an extension of PRISM that allows analyzing parametric finite state Markov chains where transition probabilities can be given as rational functions over a set of parameters.

2.5.3 Deductive Methods

Testing methods and model checking are both unsound techniques since they can not cope with infinite state spaces in finite time. Mathematical formalism has been advo-

cated as a solution to scale the analysis of programs so that the infinite nature of the search space can be manipulated symbolically. Deductive methods are based on a rigorous mathematical representation of programs semantics and allows proving properties about their dynamic evolution.

Take for example the famous Floyd-Hoare logic for proving imperative programs. It consists in a set of basic axioms and inference rules that formalize how the state of a program changes when executing statements. By using these rules and by following the syntax of the program these rules, a proof is built describing formally and symbolically the executions of the program. The construction of the proof is generally done using a proof assistant, such as Coq or Isabelle/HOL. While these tools are very helpful for verifying the correctness of a proof, the construction process remains almost manual and very painful for complex programs.

ARM4HOL. In the context of wireless embedded networks, very few verification tools are based on deductive methods. Duan and Regehr [DR10] proposed a proof framework for verifying device drivers for the ARM architecture. The proof was manually developed within the Isabelle/HOL interactive theorem prover. The framework is built around an abstract device model that proposes a modular architecture for formalizing the different changes of the hardware state resulting from driver interactions. An instantiation of this framework for the UART peripheral of an LPC2129 processor was presented. Initially, the C implementation of the device driver was compiled into ARM assembly language. By using the HOL4 formal model of ARMv4 instruction set [Fox03], and by integrating the device model of the UART peripheral, the authors constructed proofs for a number of interesting functional and timing-related properties. Naturally, the proof required substantial manual effort.

2.5.4 Abstract Interpretation

Abstract interpretation is a formal automatic method for analyzing complex dynamic systems without human intervention. As precise answers about program semantics can not be provided in general, approximation is the solely available solution. Abstract Interpretation is a theory that formalizes this notion of approximation [CC77]. The basic idea is to abstract symbolically the states of the program and to define how they are transformed by statements, in a similar way to the axioms of deductive methods. The difference lies principally in the processing of loops. Indeed, while deductive methods require the user to provide an invariant for loops in order to avoid iterating indefinitely, abstract interpretation provides extrapolation methods for inferring loops invariants without actually preforming all iterations.

Put differently, abstract interpretation proposes a general framework for (*i*) handling computable approximations of (possibly infinite) sets and (*ii*) building efficient operators that describe how these approximations evolve in a dynamic system. As a consequence, we obtain a computable abstraction of program behaviors that covers, as illustrated in Figure 2.5(c), all possible executions. This over-approximation principle constitutes the cornerstone soundness feature of this theory that guarantees that no behavior is ignored.

Stack bound. Regehr et al. [RRW03] developed the first abstract interpreter for TinyOS programs. The main purpose of the analyzer is to obtain a safe bound on the memory usage of the program. The importance of this is information is twofold. Firstly, programmers can prove that their program is stack-safe so that it will not use more memory than available. Also, it gives a quantitative information that can be useful for comparing implementations in terms of memory optimizations.

MCSquare. Another analyzer was proposed by Brauer et al. [BNS10] that operates at a lower level and analyzes binary programs for the ATmega16 MCU platform. The analysis handles partially the existence of interrupts, by omitting the case of nested occurrences, and targets solely language runtime errors.

2.6 Discussion

We summarize in Table 2.1 the existing verification solutions for the analysis of wireless embedded programs. We compare them in terms of (i) automation, (ii) coverage level of the search space and (iii) types of reported alarms. We also list the types of verified properties.

As described earlier, only deductive methods and abstract interpretation can cope with infinite search spaces with a 100% coverage. Therefore, no bug is missed which is fundamental in critical systems in order to certify the correctness of their programs. The shortcoming of these approaches is the possible false alarms generated, which is a natural and intrinsic consequence of over-approximations necessary to handle infinite spaces by a computer. That being said, these false alarms can be drastically reduced to very few cases by carefully designing abstractions as we will see in the next chapters. On the other hand, unsound approaches (testing and model checking) can not guarantee full coverage and may miss bugs, so they are inadequate for program certification. However, they are helpful for bug finding since they use under-approximations only and can report real alarms. As a first conclusion, we can say that abstract interpretation is

Table 2.1: Comparison between various verification tools for wireless embedded programs in terms of automation, coverage (○: poor, ◐: medium, ●: full) and alarms type (🔔: only genuine alarms, 🔔: possible false alarms).

Technique	Automatic	Coverage	Alarms	Safety		Performance	
				Tool	Properties	Tool	Properties
Testing	✔	○	🔔	RID	RTE	PM	Throughput
				Safe TinyOS	RTE		
				Temporal monitors	RTE, LTL		
Model checking	✔	◐	🔔	T-Check	Network	PRISM	Throughput, Energy
				KleeNet	Network		
				Anquiro	Network		
				TOS2Cprover	RTE, Assertions		
				i-CBMC	RTE, Assertions		
Deductive methods	✘	●	🔔	ARM4HOL	Hardware		
Abstract interpretation	✔	●	🔔	MCSquare	RTE	StackBound	Memory

a very adequate candidate for analyzing programs due to its automation, full coverage and adjustable precision.

Let us now discuss the types of properties analyzed by existing solutions. Most of safety verification tools focus on RTE and network properties. Hardware errors have received less attention, although device drivers represent a core component in any embedded software stack and any error in them can have extremely dangerous effects. On the other hand, for the quantitative part, we can notice that the assessment of energy and communication metrics is only provided by unsound tools. This implies that the returned results may have high confidence levels, but without any formal guarantee.

Summary. To sum up, we can say that sound approaches have not yet been fully explored for the analysis of wireless embedded software. In particular, no existing sound solution can certify the correctness of device drivers with respect to its functional properties. Similarly, quantitative analysis area is lacking of sound solutions providing guaranteed bounds of energy and communication performance metrics. We think that these two research directions are important for the wireless embedded community in particular and for the program analysis community in general. As an underlying and common framework, we propose to use the theory of abstract interpretation for its soundness, automation and adjustable precision.

2.7 Conclusion

In this chapter, we presented the state of the art of the verification solutions proposed for the particular context of wireless embedded networks. We discussed the software

architecture of these systems and how it can be flawed by programming errors. Through a high level overview of the proposed solutions, we concluded that the analysis of hardware errors and quantitative metrics represent two interesting problem not covered earlier by research community. We selected the theory of abstract interpretation as the most appropriate method for such verification and we discussed the reasons behind this choice. In the next chapter, we explain in more details this theory and how it can be used for verifying programs.

Chapter 3

Abstract Interpretation

In 1977, Patrick and Radhia Cousot published their seminal paper titled "*Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*". They described a novel formal method to deal with the undecidability of program semantics by proposing a unified framework for approximating large (and possibly) infinite mathematical constructs and also for building efficient algorithms that manipulate them. Their work relies on a rigorous mathematical formalism, that find roots in set theory, Galois connections and fixpoint computations. However, the real success of the theory has emerged after 25 years of research and improvements, when their team constructed the first abstract interpreter that was able to analyze complex industrial programs for proving the absence of runtime errors *without any false positive*. Since then, abstract interpretation has gained great notoriety within the verification community and is considered as the *de facto* theory for sound static program analysis in both academia and industry.

In this chapter, we provide a detailed description of the most important facets of this theory. We briefly introduce the underlying mathematical notions that are fundamental to understand the theory. Afterward, we explain how an abstract interpreter works *under the hood* by presenting the design process of a small analyzer.

3.1 Preliminaries

Abstract interpretation enjoys a solid and rigorous mathematical foundation that helps constructing static analyzers with provable sound results. In this section, we explain some of these notions by defining the concepts of *lattices*, *fixpoints*, *Galois connections* and *widening operators*. The reader is referred to the thesis of Patrick Cousout [Cou78] for a thorough description of these concepts. A more readable introduction to the

theory can also be found in the thesis of Antoine Miné [Min04], from which we have taken most of the following definitions.

3.1.1 Lattices

A lattice is a mathematical construct that allows (i) comparing between elements and (ii) combining them, which represent very essential operations for analyzing programs. Indeed, comparing between intermediate results is often necessary in order to ensure that the analysis effectively goes forward and new states are discovered. In addition, we also need to merge results coming from different paths in order to collect the behaviors of the programs discovered during the analysis.

Formally, the definition of a lattice $\mathcal{D}$ is based on the concept of *partially ordered sets* (posets), that defines a set with a binary relation $\sqsubseteq$ that is reflexive, transitive and anti-symmetric. In addition, to be a lattice, $\mathcal{D}$ should also verify that each element $x, y \in \mathcal{D}$ has a least upper bound, denoted $a \sqcup b$, and a greatest lower bound, denoted $a \sqcap b$. In other words, $a \sqcup b$ represents the smallest element (on the sense of $\sqsubseteq$) greater than a and b, and thus verifying three conditions: (i) $a \sqsubseteq (a \sqcup b)$, (ii) $b \sqsubseteq (a \sqcup b)$ and (iii) $\forall c \in \mathcal{D} : a \sqsubseteq c \wedge b \sqsubseteq c \Rightarrow (a \sqcup b) \sqsubseteq c$. The same explanation holds dually for $a \sqcap b$. In addition, if any set $A \subseteq \mathcal{D}$ has both a least upper bound, denoted $\bigsqcup A$, and a greatest lower bound, denoted $\bigsqcap A$, then lattice $\mathcal{D}$ is said to be *complete*. A complete lattice has therefore a least element $\bot \triangleq \bigsqcup \emptyset$ and a greatest element $\top \triangleq \bigsqcup \mathcal{D}$.

Lattices are not sufficient to formalize the executions of a program since they allow us to describe only the *structure* of the state space. To model the notion of *execution* performed by the statements of a program, we need to define the notion of *operators*. An operator $F \in \mathcal{D} \to \mathcal{D}$ is a function from a lattice to itself, and consequently describes how a statement transforms the states of the program. There exist many interesting properties about operators and we give herein the most important ones. F is said to be *monotonic* if $\forall a, b \in \mathcal{D} : a \sqsubseteq b \Rightarrow F(a) \sqsubseteq F(b)$. More generally, if F preserves existing least upper bounds of joins, i.e. $F\left(\bigsqcup A\right) = \bigsqcup\{F(a) \mid a \in A\}$, it is said to be a *complete $\sqcup$-morphism*. A particular case is when A is an increasing chain $(X_i)_{i \in I}$ (i.e. $i \leq j \Rightarrow X_i \sqsubseteq X_j$), F is said to be *continuous* if $F\left(\bigsqcup\{X_i \mid i \in I\}\right) = \bigsqcup\{F(X_i) \mid i \in I\}$.

3.1.2 Fixpoints

In program analysis, statement execution is translated to operator application in a way to mimic the flow of the program. Whilst in most cases this is reduced to function composition and lattice joins, loop statements are processed in a particular way. This is due to the fact that the control flow generated by loops can be dynamic, in the sense

that the number of iterations can depend on the result of previous computations. Such particular form of execution is formalized by the concept of *fixpoints*, that is, an element $X \in \mathcal{D}$ satisfying $F(X) = X$. In addition, a *pre-fixpoint* X is such that $X \sqsubseteq F(X)$ and a *post-fixpoint* X satisfies $X \sqsupseteq F(X)$.

As we will describe later, fixpoints allow us to collect all possible executions that occur during the iterations of a loop statement. It is important to note that an operator can have multiple fixpoints, so we denote by $\mathsf{lfp}_X\, F$ the least fixpoint of F that is greater than X and by $\mathsf{gfp}_X\, F$ the greatest fixpoint of F that is smaller than X. More particularly, we define $\mathsf{lfp}\, F \triangleq \mathsf{lfp}_\bot\, F$ and $\mathsf{gfp}\, F \triangleq \mathsf{gfp}_\top\, F$. There exist two important results that formalize the conditions about the existence of fixpoints and the way to construct them. The first one was proposed by Tarski [Tar55] and allows us to derive the existence of fixpoint solutions from the fact that the operator is monotonic and the lattice is complete:

Theorem 3.1. *The set of fixpoints of a monotonic operator F in a complete lattice is a complete lattice. Moreover, $\mathit{lfp}_X\, F = \bigsqcap\{Y \mid F(Y) \sqsubseteq Y \wedge X \sqsubseteq Y\}$, and dually, $\mathit{gfp}_X\, F = \bigsqcup\{Y \mid Y \sqsubseteq F(Y) \wedge Y \sqsubseteq X\}$.*

The second theorem, originally proposed by Kleene and later enhanced by Cousot [Cou78], describes how to solve fixpoint equations of monotonic operators:

Theorem 3.2. *Let F be a monotonic operator in a poset verifying that every increasing chain of elements $(X_i)_{i\in I}$ has a least upper bound $\bigsqcup_{i\in I} X_i$, and let $X \sqsubseteq F(X)$. Then, $F^i(X)$ is stationary at some ordinal ϵ and $\mathit{lfp}_X\, F = F^\epsilon(X) = \bigsqcup_{i<\epsilon} F^i(X)$. Moreover, if F is continuous, then $\mathit{lfp}_X\, F = \bigsqcup_{i\geq 0} F^i(X)$.*

Using these notions of lattices, operators and fixpoints, we can define the entire concrete semantics of any programming language, that is, the set of all possible executions of any program. Since this semantics is not computable, abstraction is required, as described in the following section.

3.1.3 Galois Connections

Representing the states of a program with all possible details is generally not possible on computers. For example, a program with 10 unsigned integer variables represented in 16 bits requires, in the worst case, more than 10^{40} tera bytes of memory for stocking the entire state space, which is unfeasible. For this reason, abstract interpretation is based on an approximated representation of states that omits some of its details. This transformation is based on the concept of *Galois connections*.

A Galois connection is defined between two posets $\langle \mathcal{D}, \sqsubseteq \rangle$ and $\langle \mathcal{D}^\sharp, \sqsubseteq^\sharp \rangle$, the first one is called the *concrete domain* and the second one is called the *abstract domain*.

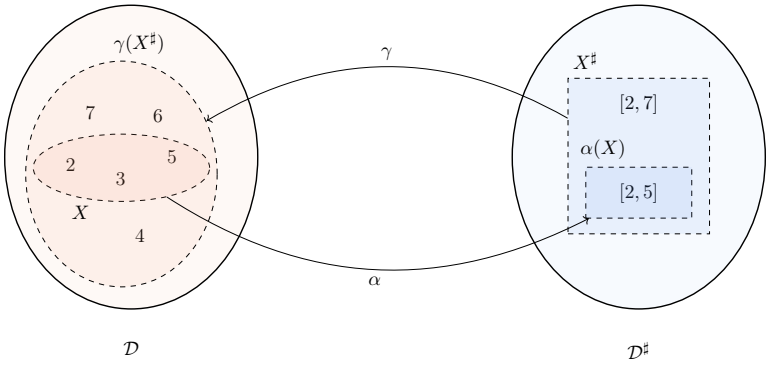


Figure 3.1: Soundness preservation of a Galois connection.

Basically, the concrete domain represents generally the real structure of the program's states whereas the abstract domain is a compressed version that does not contain all the details but has the advantage of being representable in a computer. The correspondence between these domains is established through a pair of functions (α, γ) defined as follows:

Definition 3.1. *The functions $\alpha \in \mathcal{D} \to \mathcal{D}^\sharp$ and $\gamma \in \mathcal{D}^\sharp \to \mathcal{D}$ form a Galois connection between $\mathcal{D}$ and $\mathcal{D}^\sharp$, denoted as $\mathcal{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$, if both α and γ are monotonic and:*

$$\forall X \in \mathcal{D}, X^\sharp \in \mathcal{D}^\sharp : \alpha(X) \sqsubseteq^\sharp X^\sharp \Leftrightarrow X \sqsubseteq \gamma(X^\sharp) \tag{3.1}$$

In such case, function α is called the abstraction function and γ the concretization function.

Understanding the meaning of condition (3.1) is essential. Intuitively, it states that when the abstraction of any element X is over approximated by an element $X^\sharp$, the concretization of $X^\sharp$ should also over-approximate X. In other words, this means that the relation $\sqsubseteq^\sharp$ allows us to infer sound over-approximations of X other than $\alpha(X)$, since the concretization of any abstraction $X^\sharp$ that over-approximates $\alpha(X)$ is also an over-approximation of X. As a consequence, no information is lost about X when manipulating any abstraction $X^\sharp$ verifying (3.1). Let us clarify this important concept through a simple example:

Example 3.1. We consider the case of the concrete domain $\mathcal{D} = \wp(\mathbb{N})$ representing all possible sets of integers. This domain being non computable, we choose to abstract it considering only the interval between the minimal and maximal elements. The abstraction function can therefore be defined as $\alpha(X) = [\min(X), \max(X)]$ and the concretization function as $\gamma([a, b]) = \{n \mid a \leq n \leq b\}$.

Figure 3.1 illustrates the soundness preservation of this Galois connection. The set of integers $X = \{2, 3, 5\}$ is abstracted as $\alpha(X) = [2, 5]$. If we over-approximate this

abstraction by a bigger interval $X^\sharp = [2, 7]$, and we go back to the concrete domain through concretization, we do not loose information about X since $\gamma(X^\sharp)$ contains it. ○

Several important results can be derived after establishing a Galois connection, such as the fact that:

$$\forall X \in \mathcal{D} : X \sqsubseteq \gamma \circ \alpha(X) \tag{3.2}$$

which means that abstraction of X always over-approximates X, and thus its properties are always preserved. These important facts represent the theoretical foundation of the soundness guarantee of abstract interpretation and allows building static analyzers that do not produce any false negative. This fundamental property of abstract interpretation relies indeed on the fact that the analysis always manipulates over-approximations of the concrete states, and consequently, no execution can be forgotten which makes it impossible for bugs not to be reported. However, in the same time, over-approximation may introduce spurious states that do not correspond to any concrete execution. In such situation, the analysis will report a false positive that requires a refinement process of the abstraction, as we will illustrate in the next chapter.

In addition to lattice approximation, operators are also approximated. In order to ensure that an abstract operator $F^\sharp \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is a sound over-approximation of its concrete dual $F \in \mathcal{D} \to \mathcal{D}$, it is sufficient to ensure that:

$$\forall X^\sharp \in \mathcal{D}^\sharp : \alpha \circ F \circ \gamma(X^\sharp) \sqsubseteq^\sharp F^\sharp(X^\sharp) \tag{3.3}$$

When there is equality, we say that $F^\sharp$ is the best abstraction of F with respect to the Galois connection $\mathcal{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$.

Unfortunately, a full Galois connection can not always be constructed, which is the case when the concrete elements do not have a unique abstraction, and therefore α can not be defined[1]. The theory of abstract interpretation proposes a relaxed formalism relying on a *concretization-only connection*, that overcomes this problem without sacrificing the soundness guarantee. In this formalism, the previous conditions of a typical Galois connection (3.1) are reduced to the monotonicity of γ, and we say that $X^\sharp \in \mathcal{D}^\sharp$ is an abstraction of $X \in \mathcal{D}$ if $\gamma(X^\sharp) \sqsupseteq X$. Also, the soundness condition of abstract operator $F^\sharp$ becomes:

$$\forall X^\sharp \in \mathcal{D}^\sharp : F \circ \gamma(X^\sharp) \sqsubseteq \gamma \circ F^\sharp(X^\sharp) \tag{3.4}$$

[1]For example, when using the polyhedra abstraction that represents a set of points as a finite conjunction of linear constraints, we can not find a unique abstraction of the points $\{(x, y) \mid x^2 + y^2 \leq 1\}$.

In the sequel, we will employ the concretization-based framework for describing abstractions and soundness conditions.

3.1.4 Fixpoints Revisited

Finally, to analyze programs, we need also to approximate fixpoints. There exists an important result presented in [Cou78] that establishes a relation between fixpoints computed in the concrete domain and the fixpoint computed in the abstract one:

Theorem 3.3. *If $\mathcal{D}$ and $\mathcal{D}^\sharp$ are complete lattices, then $\mathsf{lfp}_{\gamma(X^\sharp)} F \sqsubseteq \gamma(\mathsf{lfp}_{X^\sharp} F^\sharp)$.*

Consequently, fixpoints computed abstractly are always sound over-approximations of the concrete ones. As $\mathcal{D}^\sharp$ is a complete lattice, Theorem 3.1 presented earlier allows us to ensure the existence of $\mathsf{lfp}_{X^\sharp} F^\sharp$. However, in certain conditions, the existence does not imply the computability since the constructive Theorem 3.2 may not converge in finite time. Indeed, it is known that the Kleene iterations converge when any strictly ascending chain $X_0^\sharp \sqsubset X_1^\sharp \sqsubset \dots$ of elements of the lattice is necessary finite. In such case, the lattice is said to satisfy the *ascending chain condition*. Finite lattices obviously satisfy this condition. However, often in program analysis, abstract domains are infinite and do not verify the ascending chain condition. To ensure the convergence in finite time of the fixpoints computations in such situations, *widening operators* have been introduced in [CC76]. Basically, a widening operator $\triangledown \in \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ defined over a lattice $\mathcal{D}^\sharp$ takes as arguments two (generally successive) results of the Kleene iteration and tries to extrapolate them in order to ensure the convergence in finite time. Formally, the widening operator should always contain its arguments (i.e. $X^\sharp \triangledown Y^\sharp \sqsupseteq X^\sharp$ and $X^\sharp \triangledown Y^\sharp \sqsupseteq Y^\sharp$), and for every chain $(X_i^\sharp)_{i\in\mathbb{N}}$, the increasing chain $(Y_i^\sharp)_{i\in\mathbb{N}}$ defined by:

$$\begin{cases} Y_0^\sharp = X_0^\sharp \\ Y_{i+1}^\sharp = Y_i^\sharp \triangledown X_{i+1}^\sharp \end{cases} \tag{3.5}$$

is stable after a finite number of steps. When these conditions are verified, the following theorem provides a constructive method for computing sound approximations of fixpoints in a finite time thanks to a widening operator:

Theorem 3.4. *If $F^\sharp$ is an abstraction of F and $\gamma(X_0^\sharp)$ is a pre-fixpoint of F, then the chain $(Y_i^\sharp)_{i\in\mathbb{N}}$ defined by:*

$$\begin{cases} Y_0^\sharp = X_0^\sharp \\ Y_{i+1}^\sharp = Y_i^\sharp \triangledown F^\sharp(Y_i^\sharp) \end{cases}$$

is stable after a finite time.

When applied to loops, this result allows an automatic inference of loop invariants without computing all iterations. In fact, this result is a cornerstone element in the

$$
\begin{array}{rcll}
Stmt & ::= & v = e & \wr\ v \in \mathcal{V}, e \in Exp \int \\
 & | & \texttt{if } (e \bowtie 0) \texttt{ \{ } s_1 \texttt{ \} else \{ } s_2 \texttt{ \}} & \wr\ s_1, s_2 \in Stmt, \bowtie \in \{\leq, <, =, \neq\} \int \\
 & | & \texttt{while } (e \bowtie 0) \texttt{ \{s\}} & \wr\ s \in Stmt \int \\
 & | & s_1; s_2 & \\
 & & & \\
Exp & ::= & i & \wr\ i \in \mathbb{Z} \int \\
 & | & v & \wr\ v \in \mathcal{V} \int \\
 & | & -e & \wr\ e \in Exp \int \\
 & | & e_0 \diamond e_1 & \wr\ e_0, e_1 \in Exp, \diamond \in \{+, -, \times, /\} \int
\end{array}
$$

Figure 3.2: Syntax of a simple imperative language.

success of abstract interpretation for the analysis of large industrial programs that may exhibit complex loop structures.

3.2 Abstract Interpretation Tutorial

After presenting the most important mathematical background of abstract interpretation, we describe in this section more practical aspects that help understanding how this theory can be used to analyze programs. To this end, we present a step-by-step tutorial of designing an abstract interpreter for a simple imperative language. We follow a presentation based on big-step semantics which allows developing static analyzers that operate by structural induction over the program syntax.

3.2.1 Language

The first step for designing any analysis is to define the syntax of the programming language. We limit the study herein to a "toy" imperative language, described in Figure 3.2 by its syntax. Our simple language supports only assignments, tests and loops, which is sufficient to explore the most important concepts of a typical abstract interpreter. We assume that the program manipulates integer variables only, denoted as $\mathcal{V}$ with values in $\mathbb{Z}$. Arithmetic expressions are limited to elementary operations such as addition, and boolean expressions consists of comparison between arithmetic expressions. Advanced features (such as function calls, pointers, `goto`s, etc.) are not supported since they require advanced analysis techniques that are out of the scope of this didactic tutorial.

3.2.2 Concrete Semantics

The syntax of a program reflects the static nature of a program. To represent its dynamic facet, we need to define the *concrete semantics*, which represents a precise mathematical description of the program executions. The concrete semantics is defined by two notions. First, we need a *concrete semantic domain* $\langle \mathcal{D}, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ that encapsulates the required data structures for representing program executions. This representation depends on the class of the properties of interest that we are analyzing. Assume we are interested in performing a reachability analysis to collect the set of all possible states reachable during execution. Therefore, we use the powerset lattice $\langle \wp(\mathcal{E}), \subseteq, \cup, \cap, \varnothing, \mathcal{E} \rangle$ where $\mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$ represents the set of environments mapping a variable to its value.

The second notion is the *concrete transfer functions* for expressing the computations performed by a program, which are formalized as lattice operators. We divide this definition into two steps: semantics of expressions and the semantics of statements.

Concrete Semantics of Expressions. First, we need to evaluate the value of arithmetic expressions in a given environment, which is given by the function $\mathbf{E}[e] \in \mathcal{D} \to \mathbb{Z}$ defined by structural induction over the syntax of expressions as follows:

$$
\begin{array}{lll}
\mathbf{E}[e]\rho = \mathsf{match}\ e\ \mathsf{with} & & \\
\quad \mid \ i & \to & i \\
\quad \mid \ v & \to & \rho(v) \\
\quad \mid \ -e' & \to & -\mathbf{E}[e']\rho \\
\quad \mid \ e_0 \diamond e_1 & \to & \mathbf{E}[e_0]\rho \diamond \mathbf{E}[e_1]\rho
\end{array}
$$

The function $\mathbf{E}$ operates over a single environment. Later, when defining the abstract semantics, we will find it useful to define a collecting version that considers a set of environments, which is given by:

$$
\mathbf{E}[\![e]\!]R \triangleq \{\mathbf{E}[e]\rho \mid \rho \in R\} \tag{3.6}
$$

Concrete Semantics of Statements. For each statement $s \in \mathit{Stmt}$, we need to define an operator $\mathbf{S}[\![s]\!] \in \mathcal{D} \to \mathcal{D}$ that reflects how s affects the semantic domain. Statements can be divided into two categories: atomic and compound. Atomic statements are elementary operations that can change the state of the program, and are limited to assignments and filters. Consequently, defining the semantics of these statements is intrinsically dependent on the structure of the concrete semantics domain. However, compound statements, such as tests and loops, are more generic and can be

expressed recursively in terms of atomic statements. Making such differentiation is important to develop generic and modular analyzers that contain a common engine for analyzing compound statements parametrized by particular definitions of the semantics of atomic statements.

The semantics of an assignment statement is given by $\mathbf{S}[\![v = e]\!]$ and operate by updating the target variable by the evaluation of the right-hand expression in all current environments. Since an environment is just a mapping between a variable and its value, we can write:

$$\begin{array}{rcl} \mathbf{S}[\![\texttt{v = e}]\!]R & = & \{\rho' \mid \exists \rho \in R : \rho'(v) = \mathbf{E}[e]\rho \wedge (\forall v' \neq v \in V : \rho(v') = \rho(v))\} \\ & \triangleq & \{\rho[v \mapsto \mathbf{E}[e]\rho] \mid \rho \in R\} \end{array}$$

The second atomic statement is filters and its semantic $\mathbf{S}[\![?(e \bowtie 0)]\!]$ is to restrict a set of environments to those verifying a given boolean constraint. It is worth noting that filters do not appear explicitly in a program as a statement, but are useful to formalize compound statements, as it will be illustrated shortly. Formally speaking, we can define this transfer function as:

$$\mathbf{S}[\![?(e \bowtie 0)]\!]R = \{\rho \in R \mid \mathbf{E}[e]\rho \bowtie 0\}$$

Compound statements (i.e. `if`, `while` and sequences) are derived by structural induction. For the statement `if (c) {s1} else {s2}`, we analyze the true-branch `s1` and the false-branch `s2` independently and we merge the results before continuing with the following statement as follows:

$$\begin{array}{l} \mathbf{S}[\![\texttt{if } (e \bowtie 0) \texttt{ s1 else s2}]\!]R = \\ \quad \text{let } R_1 = \mathbf{S}[\![s1]\!] \circ \mathbf{S}[\![?(e \bowtie 0)]\!]R \text{ in} \\ \quad \text{let } R_2 = \mathbf{S}[\![s2]\!] \circ \mathbf{S}[\![?(e \not\bowtie 0)]\!]R \text{ in} \\ \quad R_1 \cup R_2 \end{array}$$

The case of statement sequence `s1; s2` is straightforward since it is reduced to simple mathematical composition, i.e. to compute $\mathbf{S}[\![\texttt{s1; s2}]\!]R$ we just call $\mathbf{S}[\![\texttt{s2}]\!]$ on the result of $\mathbf{S}[\![\texttt{s1}]\!]R$ as follows:

$$\mathbf{S}[\![\texttt{s1; s2}]\!]R = \mathbf{S}[\![s2]\!] \circ \mathbf{S}[\![s1]\!]R$$

The case of loops is, however, more complex since it requires handling a possibly unbounded number of iterations. Indeed, the semantics of a loop `while` $(e \bowtie 0)$ `{s}` is to repetitively execute the statement `s` until the condition is not verified. If we denote

by R the set of initial environments that reach the `while` loop, we can formulate the set of reachable states during all iterations as the fixpoint of the function:

$$F(X) = R \cup \mathbf{S}[\![\mathtt{s}]\!] \circ \mathbf{S}[\![?(e \bowtie 0)]\!]X$$

that collects the results of one iteration with the initial set of states. Finding the fixpoint $F(X) = X$ means that iterating of the environments X will not create new executions paths, and consequently, all possible traces of the loop are covered. This allows us to derive the set of environments reachable after the loop execution as:

$$\mathbf{S}[\![\mathtt{while\ (c)\ s}]\!]R = \mathbf{S}[\![?(e \not\bowtie 0)]\!](\mathsf{lfp}\, F)$$

Since F is monotonic and the concrete domain is a complete lattice, Theorems 3.1 and 3.2 allow us to conclude that the fixpoint $\mathsf{lfp}\, F$ exists and that it can be computed as supremum of the sequence $\{F^n(\bot) \mid n \in \mathbb{N}\}$. In other words, to compute this fixpoint, we iteratively build the sequence $X_{n+1} = R \cup \mathbf{S}[\![\mathtt{s}]\!] \circ \mathbf{S}[\![?(e \bowtie 0)]\!]X_n$ until $X_{n+1} = X_n$, where $X_0 = \bot$. As explained earlier, the obtained limit corresponds to a loop invariant, that is, a property satisfied at every loop iteration.

3.2.3 Abstract Semantics

Unfortunately, the concrete semantics of our simple programming language contains too much details and is therefore non-computable. Two reasons make such problem intractable. First, an element of the semantic domain $\mathcal{D}$ can be extremely large and can not be stocked in a computer memory. Second, fixpoint computations, performed during loop analysis, can require an infinite number of iterations and therefore may prevent the analysis to return a result in a finite time.

In abstract interpretation, we choose to approximate the elements of $\mathcal{D}$ by an *abstract semantic domain* $\langle \mathcal{D}^\sharp, \sqsubseteq_\sharp, \sqcup_\sharp, \sqcap_\sharp, \bot_\sharp, \top_\sharp, \rangle$ the elements of which are more compact and provide a summary of the elements of $\mathcal{D}$ by ignoring some of their details. This approximation relationship is formalized through the concretization function $\gamma \in \mathcal{D}^\sharp \to \mathcal{D}$. Abstract interpretation enjoys a rich library of abstract domains for program analysis that offer different precision granularities, and consequently, different performance tradeoffs. The most simple abstraction, albeit often practical, is the *box domain* [CC77] that can infer sound over-approximations of the values of every variable by keeping only the upper and lower bound of all possible values.

To build an abstract interpreter using the box abstract domain, we start by defining the underlying representation and lattice-related operations of this abstraction. To do so, we first define the *interval lattice* $\langle \mathcal{I}, \sqsubseteq_\mathcal{I}, \sqcup_\mathcal{I}, \sqcap_\mathcal{I}, \bot_\mathcal{I}, \top_\mathcal{I} \rangle$. An element of $\mathcal{I}$, denoted

by $[a,b]^\sharp$ where $(a,b) \in \mathbb{Z} \cup \{-\infty\} \times \mathbb{Z} \cup \{+\infty\}$, represents a (possibly unbounded) interval of integer values. Its partial order is provided by the relation $\sqsubseteq_{\mathcal{I}}$ verifying: $[a,b] \sqsubseteq_{\mathcal{I}} [c,d] \Leftrightarrow c \leq a \wedge b \leq d$. The least element is $\bot_{\mathcal{I}} \triangleq [1,-1]$ and the greatest element is $\top_{\mathcal{I}} \triangleq [-\infty,+\infty]$. The join and meet operations are given by:

$$[a,b] \sqcup_{\mathcal{I}} [c,d] \triangleq \begin{cases} [a,b] & \text{if } [c,d] = \bot_{\mathcal{I}} \\ [c,d] & \text{if } [a,b] = \bot_{\mathcal{I}} \\ [\min(a,c), \max(b,d)] & \text{otherwise} \end{cases}$$

and

$$[a,b] \sqcap_{\mathcal{I}} [c,d] \triangleq \begin{cases} [\max(a,c), \min(b,d)] & \text{if } \max(a,c) \leq \min(b,d) \\ \bot_{\mathcal{I}} & \text{otherwise} \end{cases}$$

And finally, the concretization function is defined as $\gamma_{\mathcal{I}}([a,b]) \triangleq \{n \in \mathbb{Z} \mid a \leq n \leq b\}$.

The box domain $\langle \mathcal{B}, \sqsubseteq_{\mathcal{B}}, \sqcup_{\mathcal{B}}, \sqcap_{\mathcal{B}}, \bot_{\mathcal{B}}, \top_{\mathcal{B}} \rangle$ is defined as the pointwise lifting $\mathcal{B} \triangleq \mathcal{V} \to \mathcal{I}$ of the interval lattice by the set of variables, which provides for every program variable its corresponding interval. Consequently, lattice-related operators and the concretization function can be easily derived as follows:

$$\begin{aligned} &X^\sharp \sqsubseteq_{\mathcal{B}} Y^\sharp \Leftrightarrow \forall v \in \mathcal{V} : X^\sharp(v) \sqsubseteq_{\mathcal{I}} Y^\sharp(v) \\ &\bot_{\mathcal{B}} \triangleq \lambda v.\ \bot_{\mathcal{I}} \\ &\top_{\mathcal{B}} \triangleq \lambda v.\ \top_{\mathcal{I}} \\ &X^\sharp \sqcup_{\mathcal{B}} Y^\sharp \triangleq \lambda v.\ X^\sharp(v) \sqcup_{\mathcal{I}} Y^\sharp(v) \\ &X^\sharp \sqcap_{\mathcal{B}} Y^\sharp \triangleq \lambda v.\ X^\sharp(v) \sqcap_{\mathcal{I}} Y^\sharp(v) \\ &\gamma_{\mathcal{B}}(X^\sharp) = \{\lambda v.\ n \mid n \in \gamma_{\mathcal{I}} \circ X^\sharp(v)\} \end{aligned}$$

The next step for building our analyzer is to define the abstract transfer functions that over-approximate their concrete duals. It is vital to ensure that every such a function preserves the soundness condition (3.4). In the next sections, we define formally some of these abstract transfer functions and we show how we can prove their sounds.

Abstract Semantics of Expressions. First, we define the abstract evaluation function $\mathbf{E}[\![e]\!]^\sharp \in \mathcal{B} \to \mathcal{I}$ that computes safe boundaries of the values of an expression e. It is important to note that designing such abstract functions should involve manipulation of solely abstract properties without invoking the concretization function, in order to ensure the efficiency of the analysis. We can define $\mathbf{E}[\![e]\!]^\sharp$ by structural induction over the syntax of e as follows:

$$
\begin{array}{l}
\mathbf{E}[\![e]\!]^\sharp X^\sharp = \mathsf{match}\, e\ \mathsf{with} \\
\quad \mid\ i \quad\quad \rightarrow\ [i, i] \\
\quad \mid\ v \quad\quad \rightarrow\ X^\sharp(v) \\
\quad \mid\ -e' \quad\quad \rightarrow\ \mathsf{let}[a, b] = \mathbf{E}[\![e']\!]^\sharp X^\sharp\ \ \mathsf{in}[-b, -a] \\
\quad \mid\ e_0 \diamond e_1 \ \rightarrow\ \mathsf{let}\,[a_0, b_0] = \mathbf{E}[\![e_0]\!]^\sharp X^\sharp\ \ \mathsf{and}[a_1, b_1] = \mathbf{E}[\![e_1]\!]^\sharp X^\sharp\ \ \mathsf{in} \\
\qquad\qquad\qquad\ [a_0 \diamond a_1, b_0 \diamond b_1]
\end{array}
\tag{3.7}
$$

Abstract Semantics of Statements

In a generic analyzer design, only atomic statements require specific abstraction while compound statements are more general and can have a unique expression despite the used abstract domain. The abstraction of assignments is straightforward as demonstrated by the following:

$$\mathbf{S}[\![v = e]\!]^\sharp X^\sharp = X^\sharp[v \mapsto \mathbf{E}[\![e]\!]^\sharp X^\sharp] \tag{3.8}$$

which means that firstly, the right-hand expression e is evaluated abstractly by $\mathbf{E}[\![.]\!]^\sharp$, and then the obtained bounds are affected to v. The reason behind this simple formalisation is the non-relational nature of the box domain where each variable is tracked independently of the other variables and, consequently, no relation between variables is preserved. This simplification, in addition to being sound, is efficient and allows the analysis to scale to large programs, albeit the precision is sometimes sacrificed. The case of filters is however more complicated and requires a backward evaluation of expressions and a top-bottom propagation of intermediate values in order to restrict the input abstract environment with respect to the condition. The reader is referred to [Min04, Cou99] for more details about this abstraction.

Compound statements have a more general abstraction that is independent from the structure of $\mathcal{B}$ as shown by the following:

$$
\begin{array}{ll}
\mathbf{S}[\![\texttt{s1; s2}]\!]^\sharp X^\sharp & = \mathbf{S}[\![\texttt{s2}]\!]^\sharp \circ \mathbf{S}[\![\texttt{s1}]\!]^\sharp X^\sharp \\
\mathbf{S}[\![\texttt{if } (e \bowtie 0) \texttt{ then s1 else s2}]\!]^\sharp X^\sharp & = \left(\mathbf{S}[\![s1]\!]^\sharp \circ \mathbf{S}[\![?(e \bowtie 0)]\!]^\sharp X^\sharp\right) \sqcup_{\mathcal{B}} \left(\mathbf{S}[\![s2]\!]^\sharp \circ \mathbf{S}[\![?(e \not\bowtie 0)]\!]^\sharp X^\sharp\right) \\
\mathbf{S}[\![\texttt{while } (e \bowtie 0) \texttt{ s}]\!]^\sharp X^\sharp & = \mathbf{S}[\![?(e \not\bowtie 0)]\!]^\sharp(\mathsf{lfp}\, \lambda X. X \triangledown_{\mathcal{B}} (X^\sharp \sqcup \mathbf{S}[\![s]\!]^\sharp \circ \mathbf{S}[\![?(e \bowtie 0)]\!]^\sharp X))
\end{array}
$$

The functions for composition and tests share the same structure and meaning of the concrete dual. The case of loops introduces the concept of widening since the lattice of the abstract domain $\mathcal{B}$ does not verify the condition of the ascending chain, and consequently may require a widening process in order to make the computation of the Kleene sequence converge in finite time. Proposed initially by Cousout and Cousot

[CC76], it can be defined as follows:

$$\nabla_{\mathcal{B}}([a_i, b_i]^\sharp, [a_j, b_j]^\sharp) = \left[\begin{cases} a_i & \text{if } a_i \leq a_j \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} b_i & \text{if } b_i \geq b_j \\ +\infty & \text{otherwise} \end{cases} \right]^\sharp$$

The principle of this operator is to put unstable bounds to infinity, where they cannot evolve anymore, so that the iteration terminates in a finite number of steps (as there are finitely many bounds to put to infinity).

3.2.4 Soundness Proofs

In this section, we show how we can prove the soundness of the proposed abstract transfer functions. For the sake of simplicity, we limit the proof to the cases of expression evaluation and assignments. The following lemma allows us to ensure the soundness of $\mathbf{E}[\![e]\!]^\sharp$ with respect to $\mathbf{E}[\![e]\!]$ so that the set of values computed abstractly contains all the values when the expression is evaluated in the corresponding concrete environments:

Lemma 3.1. $\forall X^\sharp \in \mathcal{B}, \forall e \in Exp : \mathbf{E}[\![e]\!] \circ \gamma_{\mathcal{B}}(X^\sharp) \subseteq \gamma_{\mathcal{I}} \circ \mathbf{E}[\![e]\!]^\sharp(X^\sharp)$

Proof. The proof is done case by case depending on the syntax of e. The simplest case is when e is a constant $i \in \mathbb{Z}$. The following steps allows to prove that the abstraction $\mathbf{E}[\![i]\!]^\sharp$, as defined in (3.7), represents a sound abstraction:

$$\begin{aligned} \mathbf{E}[\![i]\!] \circ \gamma_{\mathcal{B}}(X^\sharp) &= \{i \mid \rho \in \gamma_{\mathcal{B}}(X^\sharp)\} \\ &= \{i\} \\ &= \gamma_{\mathcal{I}}([i, i]^\sharp) \\ &= \gamma_{\mathcal{I}} \circ \mathbf{E}[\![i]\!]^\sharp(X^\sharp) \end{aligned}$$

The second case is when the expression e is a variable $v \in \mathcal{V}$. To prove the soundness, of the $\mathbf{E}[\![v]\!]^\sharp$, we proceed as follows:

$$\begin{aligned} \mathbf{E}[\![v]\!] \circ \gamma_{\mathcal{B}}(X^\sharp) &= \{\rho(v) \mid \rho \in \gamma_{\mathcal{B}}(X^\sharp)\} \\ &= \{\rho(v) \mid \rho \in \{\lambda v'.\ n \mid n \in \gamma_{\mathcal{I}} \circ X^\sharp(v')\}\} \\ &= \gamma_{\mathcal{I}} \circ X^\sharp(v) \\ &= \gamma_{\mathcal{I}} \circ \mathbf{E}[\![v]\!]^\sharp(X^\sharp) \end{aligned}$$

The third case of the expression e corresponds to the negation $-e'$ where $e' \in Exp$. For sake of clarity, we define the operator $\boxminus(R) = \{-i \mid i \in R\}$. Consequently, we can

write:

$$\begin{aligned}\mathbf{E}[\![-e']\!] \circ \gamma_{\mathcal{B}}(X^\sharp) &= \{-\mathbf{E}[e]\rho \mid \rho \in \gamma_{\mathcal{B}}(X^\sharp)\} \\ &= \boxminus(\{\mathbf{E}[e]\rho \mid \rho \in \gamma_{\mathcal{B}}(X^\sharp)\}) \\ &= \boxminus \circ \mathbf{E}[\![e']\!] \circ \gamma_{\mathcal{B}}(X^\sharp)\end{aligned}$$

By induction hypothesis and (obvious) monotony of $\boxminus$, we can derive:

$$\begin{aligned}\mathbf{E}[\![-e']\!] \circ \gamma_{\mathcal{B}}(X^\sharp) &\subseteq \boxminus \circ \gamma_{\mathcal{I}} \circ \mathbf{E}[\![e']\!]^\sharp(X^\sharp) \\ &\quad \wr \textit{ By defining } [a,b]^\sharp = \mathbf{E}[\![e']\!]^\sharp(X^\sharp) \textit{ } \int \\ &= \boxminus(\{n \mid a \leq n \leq b\}) \\ &= \{-n \mid a \leq n \leq b\} \\ &= \{n \mid -b \leq n \leq -a\} \\ &= \gamma_{\mathcal{I}}([-b,-a]^\sharp) \\ &= \gamma_{\mathcal{I}} \circ \mathbf{E}[\![e]\!]^\sharp(X^\sharp)\end{aligned}$$

Finally, the last case is the addition of two expressions $e_0 + e_1$. Here also, we define the operator $\oplus(R, R') = \{i + i' \mid i \in R \wedge i' \in R'\}$ so we get:

$$\begin{aligned}\mathbf{E}[\![e_0 + e_1]\!] \circ \gamma_{\mathcal{B}}(X^\sharp) &= \{\mathbf{E}[e_0]\rho + \mathbf{E}[e_1]\rho \mid \rho \in \gamma_{\mathcal{B}}(X^\sharp)\} \\ &\subseteq \{\mathbf{E}[e_0]\rho + \mathbf{E}[e_1]\rho' \mid \rho, \rho' \in \gamma_{\mathcal{B}}(X^\sharp)\} \\ &= \oplus(\{\mathbf{E}[e_0]\rho \mid \rho \in \gamma_{\mathcal{B}}(X^\sharp)\}, \{\mathbf{E}[e_1]\rho \mid \rho \in \gamma_{\mathcal{B}}(X^\sharp)\}) \\ &= \oplus(\mathbf{E}[\![e_0]\!] \circ \gamma_{\mathcal{B}}(X^\sharp), \mathbf{E}[\![e_1]\!] \circ \gamma_{\mathcal{B}}(X^\sharp))\end{aligned}$$

The induction hypothesis allows us to obtain:

$$\begin{aligned}\mathbf{E}[\![e_0 + e_1]\!] \circ \gamma_{\mathcal{B}}(X^\sharp) &\subseteq \oplus(\gamma_{\mathcal{I}} \circ \mathbf{E}[\![e_0]\!]^\sharp(X^\sharp), \gamma_{\mathcal{I}} \circ \mathbf{E}[\![e_1]\!]^\sharp(X^\sharp)) \\ &\quad \wr \textit{ By defining } [a_0,b_0]^\sharp = \mathbf{E}[\![e_0]\!]^\sharp(X^\sharp) \textit{ and } [a_1,b_1]^\sharp = \mathbf{E}[\![e_1]\!]^\sharp(X^\sharp) \textit{ } \int \\ &= \oplus(\{n \mid a_0 \leq n \leq b_0\}, \{n \mid a_1 \leq n \leq b_1\}) \\ &= \{n_0 + n_1 \mid a_0 \leq n_0 \leq b_0 \wedge a_1 \leq n_1 \leq b_1\} \\ &= \{n \mid a_0 + a_1 \leq n \leq b_0 + b_1\} \\ &= \gamma_{\mathcal{I}}([a_0 + a_1, b_0 + b_1]^\sharp) \\ &= \gamma_{\mathcal{I}} \circ \mathbf{E}[\![e_0 + e_1]\!]^\sharp(X^\sharp)\end{aligned}$$

□

Now, after proving the soundness of abstract arithmetic evaluation, we can give the following lemma that ensures the soundness of the assignment abstract transfer function (3.8):

Lemma 3.2. $\forall X^\sharp \in \mathcal{B} : \mathbf{S}[\![v = e]\!] \circ \gamma_{\mathcal{B}}(X^\sharp) \subseteq \gamma_{\mathcal{B}} \circ \mathbf{S}[\![v = e]\!]^\sharp(X^\sharp)$

Proof. We have:

$$\begin{aligned}
\mathbf{S}[\![v = e]\!] \circ \gamma_{\mathcal{B}}(X^\sharp) &= \{\rho[v \leftarrow \mathbf{E}[e]\rho] \mid \rho \in \gamma_{\mathcal{B}}(X^\sharp)\} \\
&\subseteq \{\rho[v \leftarrow \mathbf{E}[e]\rho'] \mid \rho, \rho' \in \gamma_{\mathcal{B}}(X^\sharp)\} \\
&= \{\rho[v \leftarrow n] \mid \rho \in \gamma_{\mathcal{B}}(X^\sharp) \wedge n \in \mathbf{E}[\![e]\!] \circ \gamma_{\mathcal{B}}(X^\sharp)\}
\end{aligned}$$

By using Lemma 3.1 we get:

$$\begin{aligned}
\mathbf{S}[\![v = e]\!] \circ \gamma_{\mathcal{B}}(X^\sharp) &\subseteq \{\rho[v \leftarrow n] \mid \rho \in \gamma_{\mathcal{B}}(X^\sharp) \wedge n \in \gamma_{\mathcal{I}} \circ \mathbf{E}[\![e]\!]^\sharp(X^\sharp)\} \\
&= \{\rho \mid \rho(v) \in \gamma_{\mathcal{I}} \circ \mathbf{E}[\![e]\!]^\sharp(X^\sharp) \wedge \forall v' \neq v : \rho(v') \in \gamma_{\mathcal{I}} \circ X^\sharp(v')\} \\
&= \{\rho \mid \forall v' : \rho(v') \in \gamma_{\mathcal{I}} \circ X^\sharp[v \rightarrow \mathbf{E}[\![e]\!]^\sharp(X^\sharp)](v')\} \\
&= \gamma_{\mathcal{B}}(X^\sharp[v \rightarrow \mathbf{E}[\![e]\!]^\sharp(X^\sharp)]) \\
&= \gamma_{\mathcal{B}} \circ \mathbf{E}[\![v = e]\!]^\sharp(X^\sharp)
\end{aligned}$$

□

3.3 Conclusion

The theory of abstract interpretation proposes a unified framework for the semantic analysis of the behaviors of discrete dynamic systems. It enjoys rigorous formal foundations that allow developing sound-by-construction static analyzers that guarantee the coverage of all possible executions.

In this chapter, we provided a detailed description of the mathematical background behind this theory along with a didactic tutorial of building an abstract interpreter from scratch. We described how we define the concrete semantics of a toy imperative language, and how we can abstract it using the box domain. Finally, we provided some soundness proofs of abstract transfer functions in order to illustrate how to ensure that a proposed abstraction fits within the requirements of the theory.

The next chapter will provide a detailed description of our static analyzer for device drivers in TinyOS, covering the formalization of a number of abstract domains tailored to this programming paradigm along with the experimental results on real-world test cases.

Chapter 4

Safety Verification of Device Drivers

In this chapter, we address the qualitative part of the thesis. We show how we can use the theory of abstract interpretation in order to analyze real-world device drivers for certifying their correctness. We provide a low-level description of important device driver operations and how these interactions are performed. These technical details allows us to illustrate the nature of functional properties we target herein. We present the necessary abstractions for performing the analysis and we discuss how to handle the problem of concurrency that hampers the verification process in general since it induces a dramatically large space of possible executions.

4.1 Introduction

The development of device drivers is an intricate process that requires particular programming skills for effectively communicate with the hardware through miscellaneous electronic protocols. Due to this particularity, device drivers are well-known to be a major cause for the instability of embedded systems, and several real-world deployments have indeed experienced such defects [BISV08, LBV06, WALJ+06]. In addition, they constitute in general a considerable fraction of the operating system and have a complex subsystem organization, which makes their manual inspection a hard task. Consequently, automatic verification tools for device drivers are extremely useful for enhancing the reliability of the system.

Motivation. As discussed previously in Section 2.6, existing safety verification tools are more focused on language runtime errors or communication protocols bugs, and can not be easily extended for driver verification for two main reasons. Firstly, they can verify only safety properties about the behavior of the program (such as assertions over variables), and do not consider properties of software/hardware interactions, which have specific aspects not present in classical program semantics, such as strobe registers[1]. The second reason is their partial support of concurrency. Indeed, existing tools consider that *interrupts* are the solely source of concurrency. However, a second concurrency is possible in device drivers that is related to hardware operations that can be performed in parallel to the execution of the program. For example, the MCU contains several subsystems that can answer the program's requests in an asynchronous way without suspending its execution. These asynchronous and parallel hardware operations affect the state of the peripheral, which in turn affects the state of the program. Therefore, it is necessary to handle this form of concurrency in order to cover all possible executions of the program.

Contributions. In this chapter, we propose a static analysis for verifying the absence of hardware interaction errors in device drivers by considering all possible executions with both forms of concurrency. Our analysis is tailored for programs running the TinyOS operating system [LMP+04], and is developed within the theory of abstract interpretation. In summary, our contributions are threefold:

- In order to find driver errors, we provide the developer with a means to express a hardware-related functional property as a special type of register automata [KF94]. This formalism can specify the pattern of correct hardware interactions for performing a particular action, along with forbidden hardware states that should be avoided.

- The property is tied to the hardware specification, not to the driver, hence it can be reused without modification to analyze different versions of a driver, or even completely different implementations of it, which we illustrate in our experimental results.

- The analysis computes a conservative over-approximation of the reachable states of the system (including program variable values and hardware state) for all possible executions. No behavior, in particular, no hardware error is omitted, which makes our analysis sound by construction and able to certify the correctness of the driver w.r.t. to the specification.

[1] A strobe register is a hardware register that does not store data but generates an action when accessed.

- To overcome the state space explosion problem due to concurrency, we do not analyze interrupt vectors at every preemption location. Instead, we proceed with a modular process that accumulates the possible preemption contexts and postpones the analysis of interrupts vectors when all these contexts are discovered.

Limitations. Since program verification is undecidable in general, our approach can suffer however from false alarms due to the over-approximations necessary to scale up. Note that other state-of-the-art formal analyzers of interrupt-based programs are generally based on bounded model checking techniques that are less vulnerable to the problem of false alarms, but can not provide guarantee about entire search space coverage and thus can suffer from false negative (*i.e.*, missing actual bugs), which makes them more adequate to bug finding than certification. That being said, in practice, our analysis can achieve a high precision level thanks to carefully designed abstractions adequate to driver verification and TinyOS semantics.

Also, we limit the description herein to drivers of the ATmega128 MCU which is a popular micro-controller found in many popular sensor platforms such as MicaZ and Waspmote. Nevertheless, the analysis is not restricted to this hardware platform and can be easily extended to other low-power architectures, such as MSP430 or ARM Cortex M0.

Outline. The remaining of the chapter is organized as follows. Section 4.2 discusses an example of a TinyOS driver, where we show also how we express a hardware functional property related to this driver. We present in Section 4.3 the assumptions and notations used in our analysis. The details of our propositions are provided in Sections 4.4 and 4.5. To simplify the presentation of our abstract interpreter, we proceed in two steps. First, we present in Section 4.4 a restricted version of our analysis limited to sequential executions where interrupt preemption is not supported. This simplification will allow us to focus on abstraction techniques dealing with the hardware state and TinyOS scheduler. After that, we extend this techniques in Section 4.5 in order to handle arbitrary interrupts preemption during execution. Experimental results of the analysis of real-world drivers are presented in Section 4.6. We end the chapter in Section 4.7 by a conclusion.

4.2 TinyOS Device Drivers

At the core of the TinyOS operating system, we find a rich library of device drivers for many microcontrollers, transceivers, sensing boards, *etc.* These programs should

encapsulate the required sequences of low-level hardware manipulations to activate the requested functionalities. The specifications of these sequences are generally described in data-sheets provided by the manufacturer of the hardware. It is vital to ensure that these *functional properties* are always preserved during runtime.

In this thesis, we choose to express these properties as a type of register automata, which we call an *Abstract Device Property* (ADP for short), that takes into account the semantics of low-level hardware interactions. An ADP is composed of a finite set of *hardware states* that corresponds to an abstract discretization of the hardware behaviors at specific moments. The dynamics between these states is modeled by a set of transitions that react to the occurrence of special low-level *events*. We can distinguish between four types of events:

Register access events. Given the set of hardware registers $\mathcal{R}$, the events $\{\mathtt{X}^{\diamond} \mid \mathtt{X} \in \mathcal{R}, \diamond \in \{r, w\}\}$ decorate transitions that model the reaction of the device when its registers are accessed by read/write statements issued by the program.

Asynchronous events. Hardware concurrency is an important concept in driver development. Many operations of the MCU sub-systems are performed independently from the program execution flow. A transition decorated with an asynchronous event, that we denote by α, allows us to model the evolution of these concurrent hardware operations.

Interrupt events. Given the set of interrupts $\mathbb{I}$, the events $\{\mathtt{int}_i \mid i \in \mathbb{I}\}$ allow the ADP to model the situations where an interrupt can occur. When a transition t is decorated with an event $\mathtt{int}_i$, the execution of the interrupt handler and the transition t are performed in a synchronous way.

Sleep event. When the TinyOS kernel terminates the execution of all posted tasks, it configures the MCU to suspend its execution waiting for interrupts. This switching between the active and inactive mode of operation is tracked by the special event `sleep`.

In addition to the occurrence of an event, each transition is decorated with a *guard*, represented as a boolean expression involving hardware registers as variables, that expresses a necessary condition for performing the transition. When both event and guard are satisfied, the ADP can move to the next state after updating the values of its registers using the *action* assignment that labels the transition.

Example 4.1. Let us take the example of the driver of CC2420, which is a low-power wireless transceiver widely used in sensor motes. It implements the IEEE 802.15.4 stan-

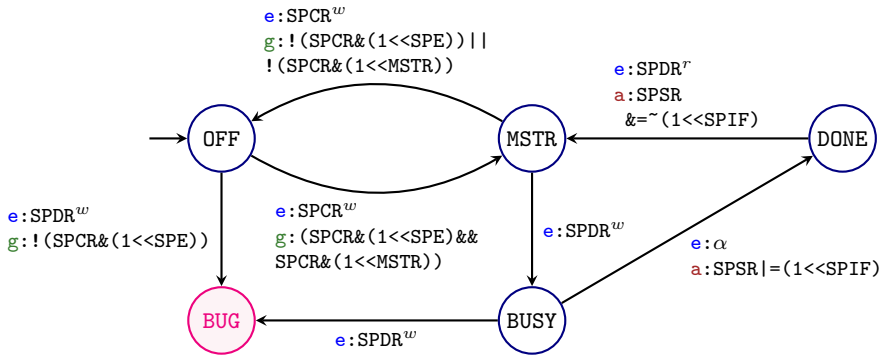


Figure 4.1: The abstract device property $\mathcal{A}_{\texttt{SPI-TX}}$ modeling a correct SPI transfer for the ATmega128 microcontroller.

dard and can be controlled via a SPI serial bus[2]. The specifications of the ATmega128 ([Atm11]) stipulate many rules to establish a correct SPI data exchange. Let us take the example of two major rules:

- No byte can be sent by the master if the bits `MSTR` and `SPE` are not set in the control register `SPCR`.
- To exchange data over the bus, the master must write into the `SPDR` data register. The transfer is handled by the MCU in an asynchronous way and therefore, the master must wait until the termination of the operation. To do so, it should continuously poll the status flag `SPIF` in the `SPSR` status register, which will be cleared by the MCU at the end of the transfer.

Figure 4.1 shows the ADP $\mathcal{A}_{\texttt{SPI-TX}}$ for the previous two SPI rules. We symbolize a transition as an arrow decorated with three fields (**e**, **g**, **a**) representing respectively the event, the guard and the action. Initially, the automaton is put in `OFF` state where data transfer is not allowed. The forbidden data transfer is modeled by a transition to the special state `BUG` decorated with the write event `SPDR`^{w} and guarded by the condition `!(SPCR&(1<<SPE))` which means that the forbidden write operation is performed when the bus is not enabled. When the program modifies the `SPE` and `MSTR` bits in the control register `SPCR`, the automaton enters the `MSTR` state. Putting data in the register `SPDR` starts the SPI communication and the bus becomes `BUSY`. During this state, no other communication can take place. The termination of the transfer is modeled with the asynchronous event α that can occur at any moment during the subsequent program execution. When this event occurs, the flag `SPIF` in the status register `SPSR` is set in order to notify the program. The latter should access the data register `SPDR`, to read

[2]SPI (*Serial Peripheral Interface*) is a serial protocol for byte exchange between devices on a shared electronic bus.

```
1  void send_spi
2   (char* data, char len){
3    char i = 0, tmp;
4    SPCR |= (1<<SPE)
5     | (1 << MSTR);
6    while (i < len) {
7      SPDR = data[i];
8      while
9       (!(SPSR & (1<<SPIF)));
10     tmp = SPDR;
11     i++;
12   }
13   SPCR &= ~(1 << SPE);
14   ...
15 }
```

(a)

```
void send_spi
 (char*data, char len){
  m_data = data;
  m_len = len;
  i = 0;
  post tx();
  SPCR |= (1<<SPE)
   | (1 << MSTR);
}
task void check(){
  if (SPSR & (1<<SPIF)){
    char tmp = SPDR;
    i++;
    post tx();
  } else
    post check();
}
```

```
task void tx(){
  if (i>=m_len){
    post end();
    return;
  }
  SPDR = m_data[i];
  post check();
}

  task end(){
    SPCR &=
      ~(1<<SPE);
    ...
  }
```

(b)

Figure 4.2: (a) Simplified driver of an SPI data exchange for the ATmega128 MCU. (b) A more complex implementation involving tasks.

the byte sent by the slave for example, which clears the `SPIF` bit and moves the ADP to `MSTR` state. ○

In Figures 4.2(a) and 4.2(b), we illustrate two driver implementations for our example functional property. The first implementation is relatively straightforward and performs an active polling on the status flag `SPIF` until termination of every byte transfer. The second implementation is more involved and exploits the tasks mechanism in order to let the scheduler execute other tasks while waiting for the end of the transfer of bytes. This driver works as follows: it starts by enabling the SPI sub-system before posting the `tx` task. The latter checks if the number of sent bytes is still less than the number of bytes to send. In this case, the next byte is written into the register `SPDR` and the task `check` is posted. This task verifies the status of the `SPIF` bit: if the bit indicates the end of the transfer, the `tx` task is posted again to send the next byte, otherwise the `check` task posts itself to continue the polling mechanism. When the last byte is sent, the task `end` is posted that turns-off the SPI sub-system. The advantage of such a procedure is that the scheduler takes control of the execution flow when the SPI is busy, which is not the case for the first implementation.

These two illustrative examples demonstrate the fact that a same functional property can be implemented with different manners and complexities. Consequently, it is necessary to analyze the *semantics* of these implementations by considering the dynamic behaviors of the program, since textual pattern matching would be inefficient to catch all possible implementations. However, the dynamic nature of programs can result in complex behaviors difficult to inspect manually. In fact, these behaviors are

not computable in general. In this work, we propose to use the theory of Abstract Interpretation to alleviate this problem. In the next sections, we propose a formulation of an abstract interpreter that takes into account, on the one hand, the specificities of TinyOS, and on the other one, the evolution of the ADP in reaction to program statements.

4.3 Assumptions and Notations

Before presenting our abstract interpretation of TinyOS device drivers, we detail the assumptions of the analysis. We assume that the input program has been preprocessed with the `ncc` compiler so that we manipulate the semantically equivalent C program. We denote by $Stmt_C$ the set of statements of this program. As a particularity of TinyOS programs, these C programs have no dynamic allocations nor function pointers. We assume also that there is no recursive functions and no backward `gotos`. Let $\mathbb{T}$ be the set of tasks and $\mathbb{I}$ the set of interrupts of the input program. The statements of these particular elements are defined by the utility function $body \in (\mathbb{T} \cup \mathbb{I}) \to Stmt_C$. Finally, we denote by $I_{ker}, I_{drv}, I_{app} \in Stmt_C$ the initialization routines for the kernel, device drivers and user applications respectively.

We formalize the ADP as a special register automaton $(\mathcal{S}, s_0, \mathcal{R}, \xi, \mathcal{T})$, where:

- $\mathcal{S}$ is the set of hardware states and s_0 is the initial state.
- $\mathcal{R}$ is the set of hardware registers.
- $\xi \triangleq \{\mathtt{X}^{\diamond} \mid \mathtt{X} \in \mathcal{R}, \diamond \in \{r, w\}\} \cup \{\mathtt{int}_i \mid i \in \mathbb{I}\} \cup \{\alpha, \mathtt{sleep}\}$ is the set of hardware events described previously in Section 4.2.
- $\mathcal{T} \subseteq \mathcal{S} \times \xi \times \mathcal{S} \times Stmt_C \times Stmt_C$ is the set of transitions where each transition $\tau = (s, e, s', g, a) \in \mathcal{T}$ moves the ADP from state s to s' whenever the event e occurs and the guard g is verified. When the transition is performed, the assignment statement a modifies the required registers.

Since we are analyzing the joint dynamics of the driver, the kernel and the hardware, the statements that affect the global state of the system are not restricted to the C atomic statements. Consequently, we consider an extended set $Stmt \triangleq Stmt_C \cup Stmt_H \cup Stmt_Q$ with the following additional statements:

- The set $Stmt_H \triangleq \{\mathsf{event}\, e \mid e \in \xi\} \cup \{\mathsf{event}^{\star}\, e \mid e \in \xi\}$ consists of the statements that trigger the ADP transitions. The statement $\mathsf{event}\, e$ fires the event e and performs a single transition of the ADP. The statement $\mathsf{event}^{\star}\, e$ is similar but

instead of making a single transition, it continues by firing all asynchronous post-transitions labeled with the event α.

- The set $Stmt_Q \triangleq \{\mathsf{dequeue}\, t \mid t \in \mathbb{T}\} \cup \{\mathsf{post}\, t \mid t \in \mathbb{T}\} \cup \{\mathsf{notask}\}$ refers to the elementary operations for manipulating the TinyOS tasks queue: removing a task from the head of the queue, posting a task at the end of the queue and testing whether the queue is empty.

4.4 Sequential Executions Analysis

In this section, we describe the design of a static reachability analysis for TinyOS programs by Abstract Interpretation. We start by presenting the analysis of sequential executions where we limit the trigger of interrupts during only sleep periods. This simplification allows us to ignore the preemption of tasks by interrupts, in order to focus on the dynamics of the program related to the interaction with the ADP and the tasks mechanism. In Section 4.5, we will extend this analysis to take into consideration the arbitrary occurrence of interrupts during execution.

4.4.1 Concrete Semantics

Our concrete semantic domain $\mathcal{D} \triangleq \wp(\mathcal{E})$ is defined as the set of subsets of concrete environments $\mathcal{E} \triangleq \mathcal{M} \times \mathcal{S} \times \mathcal{Q}$ the elements of which provide a complete characterization of the state of the system at a given program location. An environment $\rho = (m, s, q) \in \mathcal{E}$ is divided into three parts describing respectively the memory, the hardware state and the tasks queue.

The *memory environment* $\mathcal{M}$ maintains the values of the program variables, as well as hardware registers that we consider as numeric variables to facilitate their manipulation in usual C expressions. We employ the cell-based representation proposed by [Min06a] to deal with complex C data structures and pointer arithmetics. A cell $c = (v, i, \tau) \in \mathcal{C} \subseteq (V \times \mathbb{N} \times T)$ is a tuple encoding an offset i within a host variable v and having a type τ. The memory environment is defined as $\mathcal{M} \triangleq (\mathcal{C} \to \mathbb{Z}) \times (\mathcal{C} \to (V \times \mathbb{N}))$ in which we distinguish between two types of cells: numeric and pointer cells. The numeric cells are mapped to numeric values which range depends on the type of the cell. The pointer cells are considered as tuples describing the target host variable and the offset, in bytes, since the beginning of the target variable. The effect of C statements on $\mathcal{M}$ is given by a set of transfer functions $\mathbf{S}[\![.]\!]_{\mathcal{M}} \in \wp(\mathcal{M}) \to \wp(\mathcal{M})$. For the case of an atomic assignment statement, $\mathbf{S}[\![\texttt{x = exp}]\!]_{\mathcal{M}}$ evaluates the right hand side expression in every input memory environment and, for every possible value of the expression,

$$\mathbf{S}[\![\mathsf{notask}]\!]R \triangleq \{(s,m,q) \in R \mid q = \varnothing_{\mathcal{Q}}\}$$
$$\mathbf{S}[\![\mathsf{post}\, t]\!]R \triangleq \{(m,s,q') \mid \exists (m,s,q) \in R : q' = q.t\}$$
$$\mathbf{S}[\![\mathsf{dequeue}\, t]\!]R \triangleq \{(s,m,q') \mid \exists (s,m,q) \in R : q = t.q'\}$$

) We assume that X is the only register occurring in exp (
$\mathbf{S}[\![\mathtt{X = exp}]\!]R \triangleq$
) Notify the ADP about the read event (
  $\mathsf{let}\ R_1 = \mathbf{S}[\![\mathsf{event}^\star\ \mathtt{X}^r]\!]R\ \mathsf{in}$
) Update the register variable with the assignment statement (
  $\mathsf{let}\ R_2 = \{(m',s,q) \mid (m,s,q) \in R_1 \wedge m' \in \mathbf{S}[\![\mathtt{X = exp}]\!]_{\mathcal{M}}(\{m\})\}\ \mathsf{in}$
) Notify the ADP about the write event (
  $\mathbf{S}[\![\mathsf{event}^\star\ \mathtt{X}^w]\!]R_2$

$$\mathbf{S}[\![\mathsf{event}^\star\, e]\!]R \triangleq \mathsf{lfp}\ \lambda X.\ \mathbf{S}[\![\mathsf{event}\, e]\!]R \cup \mathbf{S}[\![\mathsf{event}\, \alpha]\!]X$$
$$\mathbf{S}[\![\mathsf{event}\, e]\!]R \triangleq \{(m',s',q) \mid \exists (m,s,q) \in R, \exists (s,e,s',g,a) \in \mathcal{T} : m' \in \mathbf{S}[\![a]\!]_{\mathcal{M}} \circ \mathbf{S}[\![?g]\!]_{\mathcal{M}}(\{m\})\}$$

Figure 4.3: Concrete transfer functions of sequential executions.

returns a new memory environment where the left-hand side target variable of the assignment has been updated. The test transfer function $\mathbf{S}[\![?\mathtt{exp}]\!]_{\mathcal{M}}$ allows filtering the input memory environments to retain only those where the expression `exp` can be evaluated to true. More details about the formalization of the complete semantics of C statements can be found in the work of [Min06a].

The *queue environment* $\mathcal{Q}$ provides information about the contents of the tasks queue. There exist two implementations of the queuing system in TinyOS. The first one employs a FIFO ordering of posted tasks with possible redundant occurrences of the same task. This implementation is the default mechanism used in version 1.x of TinyOS. The second implementation considers also a FIFO queue but with the restriction that the queue can not contain two entries for the same task. That is, when a task is posted again before consuming it, the queue is not modified and the second post is ignored. This behavior was chosen for TinyOS version 2.x. In the sequel of this chapter, we will describe our analysis using the first implementation, since it is more general and the second one can be easily derived from it. Nevertheless, we will provide in Section 4.6 the experimental results when using both implementations in order to give an overview about the impact of those strategies on the analysis.

Formally, we define the tasks queue environment as $\mathcal{Q} \triangleq \bigcup_{i \geq 0} \mathbb{T}^i$, where $\mathbb{T}^0 \triangleq \{\varnothing_{\mathcal{Q}}\}$ represents the singleton set of the empty queue $\varnothing_{\mathcal{Q}}$ and $\mathbb{T}^i \triangleq [0, i-1] \to \mathbb{T}$ is the set of tasks sequences $t_0 \ldots t_{i-1}$ of length i. We will employ the ordinary concatenation operator $q \cdot t$ (resp. $t \cdot q$) to denote a queue ending (resp. starting) with the task t. In addition, we introduce an auxiliary function $\mathtt{count} \in \mathcal{Q} \times \mathbb{T} \to \mathbb{N}$ giving the number of occurrences of a task in a queue.

We present in Figure 4.3 a summary of the most important transfer functions $\mathbf{S}[\![.]\!] \in \wp(\mathcal{E}) \to \wp(\mathcal{E})$ related to hardware state manipulation and TinyOS tasks. The functions $\mathbf{S}[\![\mathsf{post}\, t]\!]$, $\mathbf{S}[\![\mathsf{notask}]\!]$ and $\mathbf{S}[\![\mathsf{dequeue}\, t]\!]$ formalizing the queuing system are straightforward and just alter the queues of the input environment without modifying memory and hardware state. However, handling the effect of hardware interactions is more complex. We give the example of the function $\mathbf{S}[\![\texttt{X = exp}]\!]$ – where `X` is a register and the expression `exp` contains a read access to the same register – because it represents a frequent pattern in device drivers. For example, it is used to modify a particular bit in a register without altering the other bits, as depicted in the SPI driver in Figure 4.2(a). To handle the eventual hardware state changes, we define the functions $\mathbf{S}[\![\mathsf{event}\, e]\!]$ and $\mathbf{S}[\![\mathsf{event}^\star\, e]\!]$ that compute the possible transitions of the ADP in response to an event $e \in \xi$. Intuitively, the function $\mathbf{S}[\![\mathsf{event}\, e]\!]$ computes the effects of the one-step transitions decorated with event e and having valid guards when evaluated in the input environments. The function $\mathbf{S}[\![\mathsf{event}^\star\, e]\!]$ computes the same transitions provided by $\mathbf{S}[\![\mathsf{event}\, e]\!]$ in addition to the subsequent asynchronous transitions decorated with the asynchronous event α. Since the hardware can perform several asynchronous transitions in response to the event e, we need to collect all possible sequences of intermediate states (of arbitrary length) that the hardware can go through during this period. This is the reason for the fixpoint formulation of $\mathbf{S}[\![\mathsf{event}^\star\, e]\!]$, which is similar to the traditional definition of a transitive closure.

Using these transfer functions, we provide in Figure 4.4 a fixpoint formulation of our first concrete interpreter restricted to the analysis of the sequential executions. The interpreter starts by initializing the kernel and the drivers, and then consuming the posted tasks. After booting the user-space applications, we use two nested fixpoint computations. The inner one consumes the posted tasks and the outer one stabilizes the effect of interrupts after firing the `sleep` event when no task is waiting.

4.4.2 Abstract Semantics

In this section, we present two abstraction levels for approximating the (non computable) semantics domain $\mathcal{D}$. The first abstraction level focuses on the dynamics of ADP and maintains precise information about the hardware states in order to detect forbidden transitions. While this abstraction is sound and covers every possible execution path, it may lack some precision in presence of complex control flows that use the tasks mechanisms. Therefore, we propose a second abstraction that refines the first one by adding partial information about the contents of the tasks queue in order to avoid inconsistent tasks ordering.

$$
\begin{array}{l}
\wr \textit{ The set of initial states } \int \\
\wr \textit{ The queue is empty and all registers are initialized to } 0 \int \\
\mathsf{let}\ R_0 = \{(m, s_0, \varnothing_{\mathcal{Q}}) \mid \\
\quad m \in \mathsf{fold}(\lambda r.\lambda R.\ \mathbf{S}[\![r = 0]\!]_{\mathcal{M}} R)\ \mathcal{M}\ \mathcal{R}\}\ \mathsf{in} \\
\wr \textit{ Initialize the kernel and the drivers } \int \\
\mathsf{let}\ R_1 = \mathbf{S}[\![I_{drv}]\!] \circ \mathbf{S}[\![I_{ker}]\!] R_0\ \mathsf{in} \\
\wr \textit{ Analyze the posted tasks until emptying the queue } \int \\
\mathsf{let}\ R_2 = \mathsf{lfp}\ \lambda R.\ R_1 \cup \bigcup_{t \in \mathbb{T}} \mathbf{S}[\![\mathit{body}(t)]\!] \circ \mathbf{S}[\![\mathsf{dequeue}\ t]\!] R\ \mathsf{in} \\
\mathsf{let}\ R_3 = \mathbf{S}[\![\mathsf{notask}]\!] R_2\ \mathsf{in} \\
\wr \textit{ Initialize the user applications } \int \\
\mathsf{let}\ R_4 = \mathbf{S}[\![I_{app}]\!] R_3\ \mathsf{in} \\
\\
\mathsf{lfp}\ \lambda R.\ (\\
\quad \wr \textit{ Analyze the posted tasks } \int \\
\quad \mathsf{let}\ R_5 = \mathsf{lfp}\ \lambda R'.\ R \cup \bigcup_{t \in \mathbb{T}} \mathbf{S}[\![\mathit{body}(t)]\!] \circ \mathbf{S}[\![\mathsf{dequeue}\ t]\!] R'\ \mathsf{in} \\
\quad \wr \textit{ Move the MCU to sleep mode when no task is posted } \int \\
\quad \mathsf{let}\ R_6 = \mathbf{S}[\![\mathsf{event}^\star\ \texttt{sleep}]\!] \circ \mathbf{S}[\![\mathsf{notask}]\!] R_5\ \mathsf{in} \\
\quad \wr \textit{ Analyze the interrupts } \int \\
\quad R_4 \cup \bigcup_{i \in \mathbb{I}} \mathbf{S}[\![\mathit{body}(i)]\!] \circ \mathbf{S}[\![\mathsf{event}^\star\ \texttt{int}_i]\!] R_6 \\
)
\end{array}
$$

Figure 4.4: Concrete interpreter for sequential executions.

Hardware State Partitioning

To properly analyze the behaviors of a device driver, two important design goals should be considered. First, it is vital to keep accurate information about the hardware state since it is a key guidance element to correctly simulate the evolution of the ADP. Consequently, losing information about hardware state – when merging environments for example – should be avoided. Also, it is necessary to preserve some relationship between the hardware state of the ADP and the values of the registers because drivers try to infer the state of the device by inquiring its registers where state information is generally encoded in a set of bits.

Therefore, our first abstraction performs a partitioning with respect to the hardware states so that memory information about different states are not merged together. In other words, we collect the reachable memory environments separately for every hardware state s of the target ADP. Since we can not keep every possible detail about these environments, we build a sound summary of them using the *memory abstraction framework* described in [Min06a] and [Min12] that can over-approximate the effect of complex C constructs on memory variables efficiently. This abstraction framework is generic and can be used with any underlying numerical domain, such as the intervals domain presented earlier or even more complex relational domains such as octagons ([Min06b]) or polyhedra ([CH78]). However, in this work, we will limit ourself to the use of the intervals domain for its simplicity and efficiency. The details of these memory

$$
\begin{array}{l}
\mathbf{S}[\![\texttt{X = exp}]\!]^\sharp_{\mathcal{S}} X \triangleq \\
\quad \mathsf{let}\ X_1 = \mathbf{S}[\![\mathsf{event}^\star\ \texttt{X}^r]\!]^\sharp_{\mathcal{S}} X\ \ \mathsf{in} \\
\quad \mathsf{let}\ X_2 = \lambda s.\ \mathbf{S}[\![\texttt{X = exp}]\!]^\sharp_{\mathcal{M}} \circ X_1(s)\ \ \mathsf{in} \\
\quad \mathbf{S}[\![\mathsf{event}^\star\ \texttt{X}^w]\!]^\sharp_{\mathcal{S}} X_2 \\
\mathbf{S}[\![\mathsf{event}^\star\ e]\!]^\sharp_{\mathcal{S}} X \triangleq \mathsf{lfp}\ \lambda X'.\ X' \triangledown_{\mathcal{S}} (\mathbf{S}[\![\mathsf{event}\ e]\!]^\sharp_{\mathcal{S}} X \sqcup_{\mathcal{S}} \mathbf{S}[\![\mathsf{event}\ \alpha]\!]^\sharp_{\mathcal{S}} X') \\
\mathbf{S}[\![\mathsf{event}\ e]\!]^\sharp_{\mathcal{S}} X \triangleq \lambda s.\ \bigsqcup_{\mathcal{M}\ (s',e,s,g,a)\in\mathcal{T}} \mathbf{S}[\![a]\!]^\sharp_{\mathcal{M}} \circ \mathbf{S}[\![?g]\!]^\sharp_{\mathcal{M}} \circ X(s')
\end{array}
$$

Figure 4.5: Abstract transfer functions for hardware state partitioning.

approximations are out of the scope of this work, so we assume that we are given an abstract memory domain $\langle \mathcal{M}^\sharp, \sqsubseteq_{\mathcal{M}}, \sqcup_{\mathcal{M}}, \bot_{\mathcal{M}} \rangle$ along with a widening operator $\triangledown_{\mathcal{M}}$, a concretization function γ_M and the abstract transfer functions $\mathbf{S}[\![.]\!]^\sharp_{\mathcal{M}}$.

The formal definition of the hardware state partitioning domain $\langle \mathcal{D}^\sharp_{\mathcal{S}}, \sqsubseteq_{\mathcal{S}}, \sqcup_{\mathcal{S}}, \bot_{\mathcal{S}} \rangle$ is given by:

$$\mathcal{D}^\sharp_S \triangleq \mathcal{S} \to \mathcal{M}^\sharp$$

with the following concretization function:

$$\gamma_{\mathcal{S}}(X) \triangleq \{(m, s, q) \mid q \in \mathcal{Q} \wedge s \in \mathcal{S} \wedge m \in \gamma_{\mathcal{M}} \circ X(s)\}$$

and all lattice and widening operators are defined pointwise.

Since the number $|\mathcal{S}|$ of hardware states is finite and generally small, this partitioning does not induce excessive computational costs. It is important to note that this abstraction forgets about the contents of the tasks queue which leads to a loss of precision. Indeed, without any information about the posted tasks, preserving the soundness condition implies that we must assume that the queue can be any element of $\mathcal{Q}$ which means that our analysis will compute the effect of every possible ordering of all existing tasks.

The most interesting transfer functions are presented in Figure 4.5. The function $\mathbf{S}[\![\mathsf{event}\ e]\!]^\sharp_{\mathcal{S}}$ computes the abstract effect of an event e on the hardware and works by collecting for every possible next state s the set of transitions $(s', e, s, g, a) \in \mathcal{T}$ going from a previous state s' to s. The abstract memory environment at the state s' is then filtered by the guard g and transformed by the hardware assignment a. The function $\mathbf{S}[\![\mathsf{event}^\star\ e]\!]^\sharp_{\mathcal{S}}$ performs a sequence of widening-based iterations to compute an over-approximation of the effect of asynchronous events after the event e. The function $\mathbf{S}[\![\texttt{X = exp}]\!]^\sharp_{\mathcal{S}}$ is based on the previous two functions to over-approximate the effect of a register assignment on both the program and hardware state. Since we do not maintain any information about the posted tasks, the functions $\mathbf{S}[\![\mathsf{post}\ t]\!]^\sharp_{\mathcal{S}}$, $\mathbf{S}[\![\mathsf{dequeue}\ t]\!]^\sharp_{\mathcal{S}}$ and $\mathbf{S}[\![\mathsf{notask}]\!]^\sharp_{\mathcal{S}}$ are defined as the identity function.

$$
\begin{array}{l}
\wr \textit{ Initial abstract state } \int \\
\mathsf{let}\ X_0 = \bot_{\mathcal{S}}[s_0 \to \mathsf{fold}\ \ (\lambda r.\ \lambda X.\ \mathbf{S}[\![r = 0]\!]^\sharp_{\mathcal{M}} X)\ \top_{\mathcal{M}}\ \mathcal{R}]\ \mathsf{in} \\
\wr \textit{ Initialize the kernel and the drivers } \int \\
\mathsf{let}\ X_1 = \mathbf{S}[\![I_{drv}]\!]^\sharp_{\mathcal{S}} \circ \mathbf{S}[\![I_{ker}]\!]^\sharp_{\mathcal{S}} X_0\ \mathsf{in} \\
\wr \textit{ Analyze the posted tasks until emptying the queue } \int \\
\mathsf{let}\ X_2 = \mathsf{lfp}\, \lambda X. \\
\quad X \triangledown_{\mathcal{S}} (X_1 \sqcup_{\mathcal{S}} \bigsqcup_{\mathcal{S}}{}_{t \in \mathbb{T}} \mathbf{S}[\![body(t)]\!]^\sharp_{\mathcal{S}} \circ \mathbf{S}[\![\mathsf{dequeue}\, t]\!]^\sharp_{\mathcal{S}} X)\ \mathsf{in} \\
\mathsf{let}\ X_3 = \mathbf{S}[\![\mathsf{notask}]\!]^\sharp_{\mathcal{S}} X_2\ \mathsf{in} \\
\wr \textit{ Initialize the user applications } \int \\
\mathsf{let}\ X_4 = \mathbf{S}[\![I_{app}]\!]^\sharp_{\mathcal{S}} X_3\ \mathsf{in} \\
\\
\mathsf{lfp}\, \lambda X.\ (\\
\quad \wr \textit{ Analyze the posted tasks } \int \\
\quad \mathsf{let}\ X_5 = \mathsf{lfp}\, \lambda X'. \\
\quad\quad X' \triangledown_{\mathcal{S}} (X \sqcup_{\mathcal{S}} \bigsqcup_{\mathcal{S}}{}_{t \in \mathbb{T}} \mathbf{S}[\![body(t)]\!]^\sharp_{\mathcal{S}} \circ \mathbf{S}[\![\mathsf{dequeue}\, t]\!]^\sharp_{\mathcal{S}} X')\ \mathsf{in} \\
\quad \mathsf{let}\ X_6 = \mathbf{S}[\![\mathsf{notask}]\!]^\sharp_{\mathcal{S}} X_5\ \mathsf{in} \\
\quad \wr \textit{ Move the MCU to sleep mode } \int \\
\quad \mathsf{let}\ X_7 = \mathbf{S}[\![\mathsf{event}^\star\ \mathtt{sleep}]\!]^\sharp_{\mathcal{S}} X_6\ \mathsf{in} \\
\quad \wr \textit{ Analyze the interrupts } \int \\
\quad X_4 \sqcup_{\mathcal{S}} \bigsqcup_{\mathcal{S}}{}_{i \in \mathbb{I}} \mathbf{S}[\![body(i)]\!]^\sharp_{\mathcal{S}} \circ \mathbf{S}[\![\mathsf{event}^\star\ \mathtt{int}_i]\!]^\sharp_{\mathcal{S}} X_7 \\
)
\end{array}
$$

Figure 4.6: Abstract interpreter for sequential executions.

The abstract version of the restricted interpreter for sequential executions is depicted in Figure 4.6. We can notice that its structure is very similar to the concrete version, with the difference of employing the widening operator in order to accelerate the convergence of the fixpoint iterations for consuming tasks and firing interrupts.

Example 4.2. To explain the intuition behind this first abstraction, let us consider again the ADP of the SPI sub-system and its driver example presented in Figure 4.1 and 4.2(a) respectively. The main steps of the analysis iterations are presented in Figure 4.7 where we use the notation $\mathcal{X}_l^i$ to denote the abstract environment at line l during iteration i.

When the execution reaches for the first time the while loop at line 6, the ADP is in state `MSTR`. After the assignment statement at line 7 modifies the `SPDR` data register, the ADP moves to state `BUSY`. Since the SPI communication is asynchronous, the ADP can change its state to `DONE` at any moment, which is expressed in the abstract state $\mathcal{X}_8^1$ by two distinct state partitions. It is important to note that the value of the status register `SPSR` is different between these two partitions. This disjunction allows the analysis to infer the correct abstract environment $\mathcal{X}_9^1$ that indicates that the ADP should be in state `DONE` after the polling loop.

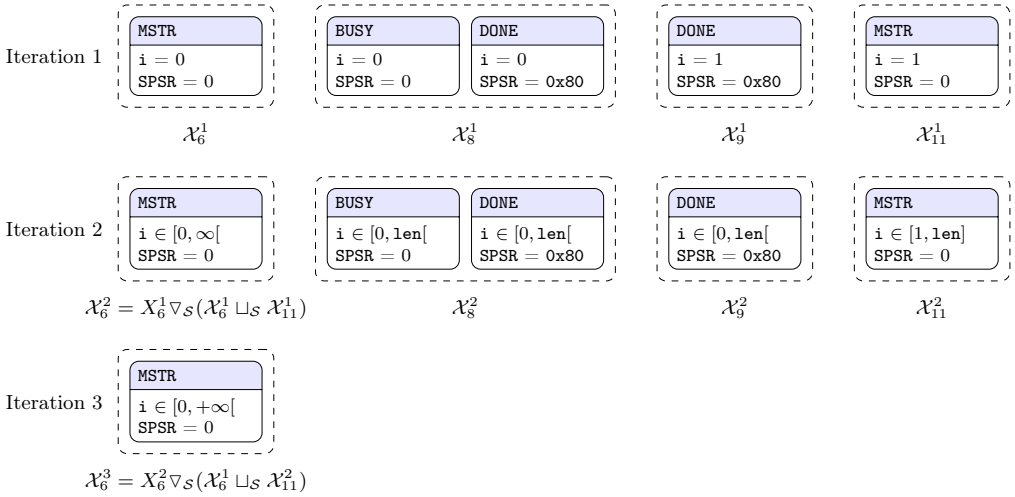


Figure 4.7: Results of fixpoint iterations obtained during the analysis of the task-less SPI driver using hardware state partitioning.

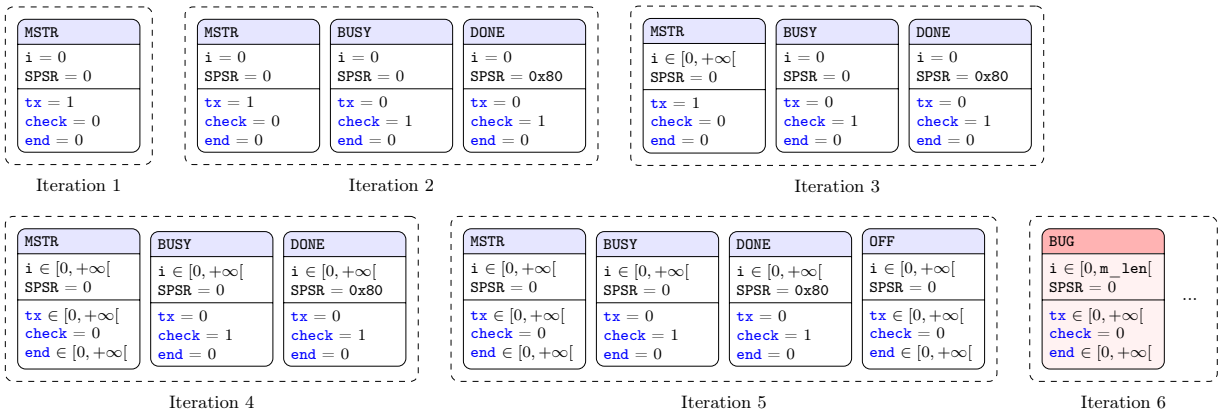


Figure 4.8: The analysis trace of a false positive during the analysis of the task-based SPI driver using the hardware state partitioning, enriched with a Parikh vector abstraction counting the number of occurrences of each task in the queue.

When performing the second fixpoint iteration of the while loop at line 6, the value of `i` is extrapolated to $[0, +\infty[$ using the widening operator. The same previous behavior is observed: the ADP moves to states `BUSY` or `DONE` depending on the termination of the transfer and the polling loop will discard the first state partition by filtering on the value of the status register `SPSR`.

At the end of the program, the `BUG` state was not reached at any step of the analysis, which constitutes a proof that the property is not violated. ○

Tasks Queue Partitioning

In some TinyOS drivers, the control flow of hardware interactions is implemented by tasks in order to free up the scheduler during polling periods. In such situations, the previous abstract domain $\mathcal{D}^\sharp_{\mathcal{S}}$ is too imprecise to reconstruct the real control flow since no information is maintained by $\mathcal{D}^\sharp_{\mathcal{S}}$ about the tasks queue. Therefore, it is necessary to refine the previous abstraction to preserve a partial view on the contents of the queue. An efficient solution is to count the number of occurrences of every task and ignore their order in the queue. This abstraction is known as a Parikh vector ([Par66]) which is generally used to approximate sets of sequences/collections of discrete objects ([Fer01b]). Unfortunately, this technique is not sufficient to prove the correctness of several task-based drivers, as we illustrate in the following example.

Example 4.3. We consider the previous task-based SPI driver presented in Figure 4.2(b). An analysis with the state partitioning domain presented in the previous section will fatally lead to the `BUG` state. Indeed, since no information is available about posted tasks, the task `tx` can be executed in the initial hardware state `OFF`, resulting in a forbidden data transfer over the deactivated SPI bus.

Even if we extend $\mathcal{D}^\sharp_{\mathcal{S}}$ with a Parikh vector, the analysis can not eliminate the false alarm. To explain further this problem, we depict in Figure 4.8 the results of the fixpoint iterations using the extended domain and we focus on the obtained abstract environments during the tasks consumption step, which corresponds to the computation of X_5 in the previous abstract interpreter. The execution trace of these iterations can be summarized as follows:

1. Initially, after executing the `send_spi` function, the task `tx` is posted to send the first byte while the ADP is in state `MSTR`.

2. During the first fixpoint iteration, the task `tx` is executed and the transfer is started, which generates two additional partitions for the states `BUSY` and `DONE`. The task `check` is posted in order to continuously check the termination of the transfer.

3. The task `check` filters its input abstract environment depending on the status flag `SPIF`. The next byte is prepared to be sent when this flag is set, which corresponds to the state partition `DONE`. Since we are using the widening operator, the byte index `i` is extrapolated to $[0, +\infty[$.

4. The execution of the task `tx` over this new abstract environment will generate two cases: the transfer of the next byte or the end of the transfer. The latter case is performed by posting the task `end`. It is important to note that this task is posted

while the ADP is in state `MSTR`. By merging this result with the previous `MSTR` partition, and by widening, we reach an imprecise and nondeterministic situation resulting from the fact that `tx` $\in [0, +\infty[$ and `end` $\in [0, +\infty[$. These values imply that no constraint is available for these two tasks, and therefore any ordering of them can happen.

5. When the analyzer executes the task `end` in this imprecise context, it will shut down the SPI, moving the ADP to state `OFF`. Due to the previous nondeterminism, the task `tx` can be executed in this hardware state, which will lead to a false positive after writing to data register `SPDR` while the bus is inactive.

○

The previous example shows that counting the number of posts can be insufficient to reconstitute a correct execution flow of tasks. More precisely, the false positive originated from a lack of relation between the entries of the Parikh vector. Indeed, for this illustrative example, there exists an exclusive-or relation between the presence of the tasks `tx` and `end` together in the queue, and providing the analyzer with such information will eliminate the false positive. To provide such relationship, we propose to build a partitioning with respect to the presence of tasks. This form of disjunction allows the analyzer to keep separate two sets of environments when a task is not posted in both cases.

Formally, we define the task partitioning abstract domain $\langle \mathcal{D}^\sharp_{\mathcal{Q}}, \sqsubseteq_{\mathcal{Q}}, \sqcup_{\mathcal{Q}}, \bot_{\mathcal{Q}} \rangle$ having the following structure:

$$\mathcal{D}^\sharp_{\mathcal{Q}} \triangleq \wp(\mathbb{T}) \to (\mathcal{D}^\sharp_{\mathcal{S}} \times (\mathbb{T} \to \mathcal{N}^\sharp))$$

where $\langle \mathcal{N}^\sharp, \sqsubseteq_{\mathcal{N}}, \sqcup_{\mathcal{N}}, \bot_{\mathcal{N}} \rangle$ is a numeric abstract domain approximating a set of integers, such as intervals, provided with its concretization function $\gamma_{\mathcal{N}}$. Basically, when $X \in \mathcal{D}^\sharp_{\mathcal{Q}}$ is an abstract environment and $T \in \wp(\mathbb{T})$ is a set of tasks, the partition $X(T)$ provides an over-approximation of the memory, hardware and queue environments when only the tasks $t \in T$ are present in the queue. To obtain the concrete environments approximated by the abstract environment X, we define the following concretization function:

$$\begin{aligned}
&\gamma_{\mathcal{Q}}(X) \triangleq \\
&\quad \{(m, s, q) \mid \exists T \in \wp(\mathbb{T}) : (m, s, -) \in \gamma_{\mathcal{S}} \circ \downarrow_S \circ X(T) \\
&\qquad \wedge \forall t \in T : \texttt{count}(q, t) \in \gamma_{\mathcal{N}} \circ \downarrow^t_Q \circ X(T) \wedge \texttt{count}(q, t) > 0 \\
&\qquad \wedge \forall t \notin T : \texttt{count}(q, t) = 0\}
\end{aligned}$$

where $\downarrow_S (X_S, X_T)$, $\downarrow_Q (X_S, X_T)$ and $\downarrow^t_Q (X_S, X_T)$ are three projection operators to retrieve respectively X_S, X_T and $X_T(t)$ from an abstract environment $(X_S, X_T) \in \mathcal{D}^\sharp_{\mathcal{Q}}$.

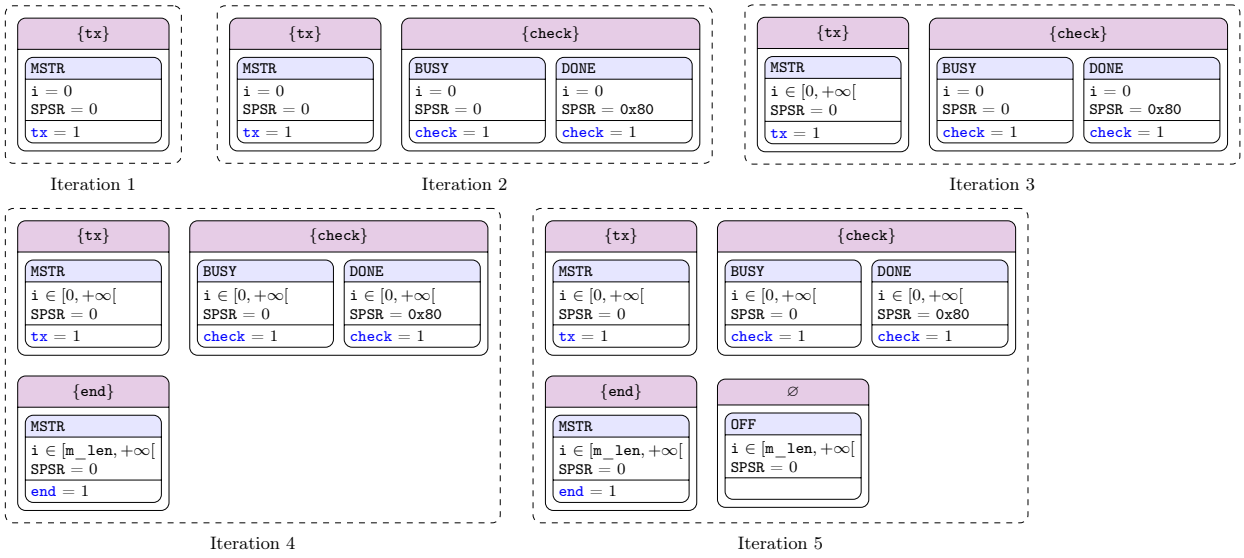


Figure 4.9: A summary of fixpoint iterations obtained during the analysis of the SPI driver using tasks queue partitioning.

Intuitively, the function $\gamma_{\mathcal{Q}}$ obtains the concrete memory and hardware environments using the previous concretization function $\gamma_{\mathcal{S}}$. The concrete queues environments are constructed using the condition that the number of occurrences of every task should be in the range of its corresponding entry in the Parikh vector.

Since the transfer functions for this domains share many similar constructs, we limit ourself to the presentation of the case of the statement post t:

$$
\begin{aligned}
&\mathbf{S}[\![\mathsf{post}\, t]\!]^\sharp_{\mathcal{Q}} X \triangleq \\
&\quad \lambda T. \begin{cases} \bot_{\mathcal{S}}, \lambda t'.\ \bot_{\mathcal{N}} \text{ if } t \notin T \\ \downarrow_{\mathcal{S}} \circ X(T), \downarrow_{\mathcal{Q}} \circ X(T)[t \leftarrow \mathtt{inc}^\sharp_{\mathcal{N}} \circ \downarrow^t_{\mathcal{Q}} \circ X(T)] \text{ otherwise} \end{cases} \\
&\quad \sqcup_{\mathcal{Q}} \\
&\quad \lambda T. \begin{cases} \bot_{\mathcal{S}}, \lambda t'.\ \bot_{\mathcal{N}} \text{ if } t \notin T \\ \downarrow_{\mathcal{S}} \circ X(T \setminus \{t\}), \downarrow_{\mathcal{Q}} \circ X(T \setminus \{t\})[t \leftarrow \mathtt{one}^\sharp_{\mathcal{N}}] \text{ otherwise} \end{cases}
\end{aligned}
$$

The first part of the union $\sqcup_{\mathcal{Q}}$ handles the case where the task t was already posted and operates by incrementing the number of its occurrences using the abstract incrementation function $\mathtt{inc}^\sharp_{\mathcal{N}}$ that verifies the soundness condition:

$$\forall X \in \mathcal{N}^\sharp : \gamma_{\mathcal{N}} \circ \mathtt{inc}^\sharp_{\mathcal{N}}(X) \supseteq \{i + 1 \mid i \in \gamma_{\mathcal{N}}(X)\}$$

The case where t was not present in the queue is handled by the second part, which updates the partitions $X(T \setminus \{t\})$ by setting the Parikh vector entry of t to the abstract element $\mathtt{one}^\sharp_{\mathcal{N}}$ that verifies the soundness condition: $1 \in \gamma_{\mathcal{N}}(\mathtt{one}^\sharp_{\mathcal{N}})$.

Example 4.4. We illustrate in Figure 4.9 the advantage of the tasks queue partitioning for proving the correctness of the previous SPI driver. The execution trace is relatively similar to the previous case for the first three iterations. During the fourth iteration, the task `end` is posted in a partition different from the task `tx`, which avoids the previous nondeterminism and shuts down the SPI bus safely. ○

4.5 Preemptive Executions Analysis

Until now, we have considered that interrupts can occur only during inactivity periods of the MCU. This assumption allowed us to simplify the presentation of the abstractions related to the hardware state and the tasks queue. In this section, we extend the previous analysis to take into consideration the preemption of execution by interrupts at any moment of the program lifetime. We define new concrete and abstract semantics, that build on the previous ones, to soundly over-approximate the set of reachable hardware states during all possible concurrent executions.

4.5.1 Concrete Semantics

To add interrupts preemption to our previous analysis, we need to care about (*i*) when an interrupt can be fired and (*ii*) when the MCU is configured to execute its corresponding interrupt vector. In this work, we focus on the second consideration and we approximate the first one using nondeterminism. In other words, as long as an interrupt is not *masked* by software, we consider that it can happen at any moment, which can be implemented as a nondeterministic choice to execute or not the interrupt handler before executing any statement. Nevertheless, the imprecision caused by the nondeterminism can be reduced by filtering the hardware states in which interrupts can not occur. This can be done by adding transitions, labeled with interrupt events $\texttt{int}_i$, that go from the filtered states to a special absorbing state that has no successor.

An interrupt can be masked at two levels: *globally* and *partially*. The first level is handled by a Global Interrupt Enable (GIE) bit found in most MCUs. For the case of the ATmega128 MCU that we are considering in this work, the I bit located at the last position in the status register `SREG` must be set in order to enable interrupts. In the following, we will denote by gcond $\triangleq$ `SREG & (1 << 7) != 0` the condition expression that verifies that the I bit is set in `SREG`. Also, we define two shortcut statements `cli` $\triangleq$ `SREG &= ~(1 << 7)` and `sei` $\triangleq$ `SREG |= (1 << 7)` to respectively clear and set the I bit.

The second masking level consists in the inhibition of a partial set of interrupts, performed generally through the configuration of particular control registers. Since

these configurations differ from an interrupt to another, we define the function icond $\in \mathbb{I} \to Stmt_C$ giving for every interrupt its corresponding firing condition, formulated as a C boolean expression. For example, to allow the occurrence of the Timer0 compare interrupt (T0CI), the following condition should be verified:

$$\mathsf{icond}(\text{T0CI}) \triangleq \begin{array}{l}\texttt{(TCCR0 \& ((1 << CS02) | (1 << CS01) |(1 << CS00)))} \\ \texttt{\&\& (TIMSK \& (1 << TOIE0))}\end{array}$$

The first condition ensures that a non null prescaler is configured in the control register `TCCR0`, otherwise no clock will source the timer sub-system. The second condition checks whether this particular interrupt is enabled in the timer mask register `TIMSK`.

Since these conditions are expressed as predicates over hardware registers – which are considered as normal program variables – we can use our previous concrete semantic domain $\mathcal{D}_{\mathcal{I}} \triangleq \mathcal{D}$ to encapsulate the mask values of interrupts. However, we need to extend the transfer functions in order to handle the nondeterministic execution of interrupt vectors when they are not masked. To do so, we only need to define the transfer functions for the atomic C statements (assignments and tests) and for the additional statements set $Stmt_H$ and $Stmt_Q$. The remaining transfer functions are kept unmodified because they are ultimately reduced, by structural induction on the syntax, to atomic statements. Let s be one of these atomic statements. We define its preemption-aware transfer function as follows:

$$\begin{array}{l}\mathbf{S}[\![s]\!]_{\mathcal{I}} R \triangleq \mathbf{S}[\![s]\!](R \cup \bigcup_{i \in \mathbb{I}} \\ \quad \mathsf{let}\ R_1 = \mathbf{S}[\![\mathsf{event}^\star\ \texttt{int}_i]\!] \circ \mathbf{S}[\![?\,\mathsf{icond}(i)]\!] \circ \mathbf{S}[\![?\,\mathsf{gcond}]\!] R\ \ \mathsf{in} \\ \quad \mathbf{S}[\![\texttt{sei}]\!] \circ \mathbf{S}[\![body(i)]\!]_{\mathcal{I}} \circ \mathbf{S}[\![\texttt{cli}]\!] R_1)\end{array}$$

Basically, this function executes the statement s over the union of the input environments R and the post-execution environments of the enabled interrupts. These environments are obtained by first filtering R in order to keep only the environments where the condition expressions of the global enable bit and the partial mask of i are true. After that, we signal the occurrence of the interrupt to the ADP by calling the function $\mathbf{S}[\![\mathsf{event}^\star\ \texttt{int}_i]\!]$. Finally, we execute the interrupt vector body by first clearing the global interrupt enable bit and then setting it again as specified by the ATmega128 data sheet.

Two important points should be noted. First, using the union of the post-interrupts environments implies that *at most* one interrupt handler is executed. This choice is justified by the fact that the MCU, after returning from an interrupt, must execute at least one instruction before allowing the execution of the next interrupt, which avoids a continuous succession of interrupts that prevents a function from terminating its

$$
\begin{array}{l}
\wr \textit{ The set of initial states } \int \\
\mathsf{let}\ R_0 = \{(m, s_0, \varnothing_{\mathcal{Q}}) \mid \\
\quad m \in \mathsf{fold}(\lambda r.\lambda R.\ \mathbf{S}[\![r = 0]\!]_{\mathcal{M}} R)\ \mathcal{M}\ \mathcal{R}\}\ \mathsf{in} \\
\wr \textit{ Initialize the kernel and the drivers } \int \\
\mathsf{let}\ R_1 = \mathbf{S}[\![I_{drv}]\!]_{\mathcal{I}} \circ \mathbf{S}[\![I_{ker}]\!]_{\mathcal{I}} R_0\ \mathsf{in} \\
\wr \textit{ Analyze tasks } \int \\
\mathsf{let}\ R_2 = \mathsf{lfp}\ \lambda R.\ R_1 \cup \bigcup_{t \in \mathbb{T}} \mathbf{S}[\![\mathit{body}(t)]\!]_{\mathcal{I}} \circ \mathbf{S}[\![\mathsf{dequeue}\ t]\!]_{\mathcal{I}} R\ \mathsf{in} \\
\mathsf{let}\ R_3 = \mathbf{S}[\![\mathsf{notask}]\!]_{\mathcal{I}} R_2\ \mathsf{in} \\
\wr \textit{ Enable interrupts and analyze the user apps initialization } \int \\
\mathsf{let}\ R_4 = \mathbf{S}[\![\texttt{oldSREG = SREG}]\!] \circ \mathbf{S}[\![I_{app}]\!]_{\mathcal{I}} \circ \mathbf{S}[\![\texttt{sei}]\!] R_3\ \mathsf{in} \\
\\
\mathsf{lfp}\ \lambda R.\ (\\
\quad \wr \textit{ Restore the global interrupts mask and analyze tasks } \int \\
\quad \mathsf{let}\ R_5 = \mathbf{S}[\![\texttt{SREG = oldSREG}]\!] R\ \mathsf{in} \\
\quad \mathsf{let}\ R_6 = \\
\quad\quad \mathsf{lfp}\ \lambda R'.\ R_5 \cup \bigcup_{t \in \mathbb{T}} \mathbf{S}[\![\mathit{body}(t)]\!]_{\mathcal{I}} \circ \mathbf{S}[\![\mathsf{dequeue}\ t]\!]_{\mathcal{I}} R'\ \mathsf{in} \\
\quad \mathsf{let}\ R_7 = \mathbf{S}[\![\mathsf{notask}]\!]_{\mathcal{I}} R_6\ \mathsf{in} \\
\quad \wr \textit{ Save the global interrupts mask and enable interrupts } \int \\
\quad \mathsf{let}\ R_8 = \mathbf{S}[\![\texttt{sei}]\!] \circ \mathbf{S}[\![\texttt{oldSREG = SREG}]\!] R_7 \\
\quad \wr \textit{ Move the MCU to sleep mode } \int \\
\quad \mathsf{let}\ R_9 = \mathbf{S}[\![\mathsf{event}^\star\ \texttt{sleep}]\!] R_7\ \mathsf{in} \\
\quad \wr \textit{ Analyze interrupts } \int \\
\quad R_4 \cup \\
\quad\quad \bigcup_{i \in \mathbb{I}} \mathbf{S}[\![\texttt{sei}]\!] \circ \mathbf{S}[\![\mathit{body}(i)]\!]_{\mathcal{I}} \circ \mathbf{S}[\![\texttt{cli}]\!] \circ \mathbf{S}[\![\mathsf{event}^\star\ \texttt{int}_i]\!] R_9 \\
)
\end{array}
$$

Figure 4.10: Concrete interpreter for preemptive executions.

computations. Second, this formulation allows the analysis of nested interrupts by using the preemptive transfer function $\mathbf{S}[\![\mathit{body}(i)]\!]_{\mathcal{I}}$. Nevertheless, since the I-bit is automatically cleared at the beginning of the interrupt, the body of the corresponding routine should execute the `sei` statement to allow this feature, which is taken into account by our semantics.

Using this preemption mechanism, we define in Figure 4.10 the concrete preemptive interpreter of TinyOS programs. It shares with the previous sequential interpreter, presented in Figure 4.4, most of its structure with two major differences. Firstly, the body of the initialization procedures, tasks and interrupts are analyzed using the preemption-aware transfer function $\mathbf{S}[\![.]\!]_{\mathcal{I}}$, instead of the sequential version $\mathbf{S}[\![.]\!]$. Secondly, we have instrumented the interpreter with statements to control the global interrupt mask as performed by the TinyOS scheduler. Note that, at specific locations, the scheduler saves the register `SREG` in a backup variable, that we denote by `oldSREG`, in order to restore it later to preserve the modifications performed by the tasks.

$$
\begin{array}{l}
\mathbf{S}[\![\texttt{SREG = exp}]\!]^\sharp_{\mathcal{K}}(G,X) \triangleq \\
\quad \mathsf{let}\ X_1 = \lambda I.\ \mathbf{S}[\![\texttt{SREG = exp}]\!]^\sharp_{\mathcal{Q}} \circ X(I)\ \mathsf{in} \\
\quad \mathsf{let}\ G_1 = \begin{cases}
\mathbf{1} & \text{if } \forall I \in \wp(\mathbb{I}) : \mathbf{S}[\![?\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) \neq \bot_{\mathcal{Q}} \wedge \mathbf{S}[\![?!\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) = \bot_{\mathcal{Q}} \\
\mathbf{0} & \text{if } \forall I \in \wp(\mathbb{I}) : \mathbf{S}[\![?\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) = \bot_{\mathcal{Q}} \wedge \mathbf{S}[\![?!\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) \neq \bot_{\mathcal{Q}} \\
\top_{\mathbf{01}} & \text{if } \forall I \in \wp(\mathbb{I}) : \mathbf{S}[\![?\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) \neq \bot_{\mathcal{Q}} \wedge \mathbf{S}[\![?!\,\mathsf{gcond}]\!]^\sharp_{\mathcal{Q}} \circ X_1(I) \neq \bot_{\mathcal{Q}} \\
\bot_{\mathbf{01}} & \text{otherwise}
\end{cases} \\
\quad \mathsf{in}(G_1, X_1)
\end{array}
$$

$$
\begin{array}{l}
\mathbf{S}[\![\texttt{X = exp}]\!]^\sharp_{\mathcal{K}}(G,X) \triangleq \\
\quad \mathsf{let}\ X_1 = \lambda I.\ \mathbf{S}[\![\texttt{X = exp}]\!]^\sharp_{\mathcal{Q}} \circ X(I)\ \mathsf{in} \\
\quad \mathsf{let}\ X_2 = \lambda I.\ \bigsqcup_{\mathcal{Q}\ i \in I, I' \in \wp(\mathbb{I})} \mathbf{S}[\![?\,\mathsf{icond}(i)]\!]^\sharp_{\mathcal{Q}} \circ X_1(I') \sqcup_{\mathcal{Q}} \bigsqcup_{\mathcal{Q}\ i \notin I, I' \in \wp(\mathbb{I})} \mathbf{S}[\![?!\,\mathsf{icond}(i)]\!]^\sharp_{\mathcal{Q}} \circ X_1(I') \\
\quad \mathsf{in}(G, X_2)
\end{array}
$$

Figure 4.11: Abstract transfer functions for the $\mathcal{D}^\sharp_{\mathcal{K}}$ domain.

4.5.2 Abstract Semantics

In this section, we develop an abstraction of the domain $\mathcal{D}_{\mathcal{I}}$ and the transfer functions $\mathbf{S}[\![.]\!]_{\mathcal{I}}$ that approximate the dynamics of preemptive executions. To do so, we divide our problem into two parts: (i) the maintenance of an abstract view about the enabled interrupts, and (ii) the computation of the effect of an enabled interrupt on the execution flow. Therefore, we start by presenting an approximation of the masking system before describing how to use this abstraction to analyze the preemptive executions of a TinyOS program.

Abstraction of Interrupt Masks

We approximate the interrupt masks by breaking the relation between the global masking level and the partial one. This separation allows us to build efficient transfer functions by sacrificing some precision. Formally, we define the abstract mask domain $\langle \mathcal{D}^\sharp_{\mathcal{K}}, \sqsubseteq_{\mathcal{K}}, \sqcup_{\mathcal{K}}, \bot_{\mathcal{K}} \rangle$ as the following product:

$$\mathcal{D}^\sharp_{\mathcal{K}} \triangleq \underbrace{\{\bot_{\mathbf{01}}, \mathbf{0}, \mathbf{1}, \top_{\mathbf{01}}\}}_{\mathcal{K}^\sharp_{\mathcal{G}}} \times \underbrace{(\wp(\mathbb{I}) \to \mathcal{D}^\sharp_{\mathcal{Q}})}_{\mathcal{K}^\sharp_{\mathcal{P}}}$$

The global mask is maintained by the lattice $\mathcal{K}^\sharp_{\mathcal{G}}$ that is identical to the powerset lattice $\wp(\{0,1\})$ and encodes all possible states of the I bit. The second masking level is provided by the lattice $\mathcal{K}^\sharp_{\mathcal{P}}$ defining a partitioning with respect to the activated interrupts. From the pointwise definition of the lattice operators of $\mathcal{K}^\sharp_{\mathcal{P}}$ and the simple definition of $\mathcal{K}^\sharp_{\mathcal{G}}$, we can easily derive the definition of the bottom element $\bot_{\mathcal{K}}$ and the

operators $\sqcup_{\mathcal{K}}$, $\sqsubseteq_{\mathcal{K}}$ and $\triangledown_{\mathcal{K}}$.

The set of concrete environments corresponding to a given abstract interrupt mask is given by the following concretization function:

$$
\begin{array}{l}
\gamma_{\mathcal{K}}(G, X) \triangleq \\
\quad \mathsf{let}\ R = \{(m,s,q) \mid \exists I \in \wp(\mathbb{I}) : (m,s,q) \in \gamma_{\mathcal{Q}} \circ X(I) \wedge \\
\qquad \forall i \in I : (m,s,q) \in \mathbf{S}[\![?\,\mathsf{icond}(i)]\!] \circ \gamma_{\mathcal{Q}} \circ X(I) \wedge \forall i \notin I : (m,s,q) \in \mathbf{S}[\![?!\,\mathsf{icond}(i)]\!] \circ \gamma_{\mathcal{Q}} \circ X(I)\} \\
\quad \mathsf{in}\ \ \mathsf{match}\ G\ \ \mathsf{with} \\
\qquad \mid \top_{\mathbf{01}} \rightarrow R \\
\qquad \mid \mathbf{1} \rightarrow \mathbf{S}[\![?\,\mathsf{gcond}]\!]R \\
\qquad \mid \mathbf{0} \rightarrow \mathbf{S}[\![?!\,\mathsf{gcond}]\!]R \\
\qquad \mid \bot_{\mathbf{01}} \rightarrow \varnothing
\end{array}
$$

Let us define now the abstract transfer functions $\mathbf{S}[\![.]\!]^{\sharp}_{\mathcal{K}}$ related to $\mathcal{D}^{\sharp}_{\mathcal{K}}$. The most important cases are presented in Figure 4.11. The transfer function $\mathbf{S}[\![\texttt{SREG = exp}]\!]^{\sharp}_{\mathcal{K}}(G, X)$ handles the change of the global enable bit I after a modification of the `SREG` status register, as performed by the two shortcut statements `sei` and `cli`. After updating each partition with the assignment statement, we check the different values of the mask expression gcond over the resulting environments and update the abstract global bit accordingly. The effect of changing the partial masks is defined by the function $\mathbf{S}[\![\texttt{X = exp}]\!]^{\sharp}_{\mathcal{K}}(G, X)$, where `X` is a hardware register different than `SREG` and present in at least one of the expressions icond. The function updates the partitions with the effect of the assignment and rebuilds the partitions again depending on the evaluation of the expressions icond.

Abstraction of Preemption

Dealing with preemption in interrupt-rich programs is a challenging task. Several approaches have been developed offering different precision/efficiency tradeoffs. Sequentialization ([Mon07, BK11]) is a simple solution consisting in instrumenting the original program with nondeterministic calls to the interrupt handlers. Since the number of execution paths may become intractable, different forms of partial order reduction are proposed to restrict the locations of this instrumentation. This method allows a precise analysis, but becomes inefficient in the presence of a large number of interrupts with possible nested occurrences. A more interesting approach, proposed by i-CBMC model checker ([KLM+15]), alleviates the need to apply partial order reductions and provides a better scalability with less instrumentation effort. It is based on the definition of a partial order on preemption traces that uses a set of logical clocks to symbolically encode the different interleavings of interrupts. Whilst this method is effective in many

test cases, it lacks the soundness guarantee and can not cover all possible execution traces of complex programs in finite time.

In our work, we aim at proposing a more efficient approach that guarantees the soundness condition and avoids executing interrupt handlers every time they are enabled. Our solution is based on the Modular Abstract Interpretation framework ([CC02]) and is similar to the approach of the static analyzer AstréeA ([Min11, Min14]). The general idea of this method consists in analyzing the *parts* of the program *separately* and then compose the local results of every part to get an aggregate view of the whole program. Since the interactions between these parts can be complex, the analysis may be iterated several times to obtain the correct results. Indeed, the initial analysis iteration has no information about the influence of a part on another and is therefore performed by assuming that there is no such interactions. However, during this iteration, the analysis can discover new interactions *on the fly*, such as new call sites, providing a more accurate view on the actual interaction map. Consequently, successive iterations are required until we ensure that all interactions have been discovered.

In our case, the high-level functions (initialization functions, tasks and interrupts) constitute the *parts* of the program with the restriction that only interrupts can preempt execution. The analysis processes each part separately and constructs two inter-parts information: the preemption contexts and the return contexts:

1. The *preemption context* of an interrupt represents the collection of the abstract environments where an interrupt may occur. It is constructed on the fly during the analysis of the other parts by computing the union of the abstract environments that verify the enable condition of the interrupt.

2. The analysis computes the *return contexts* of interrupts by executing separately each interrupt handler over its corresponding preemption context. It will be used during the next iteration to soundly *emulate* the execution of the interrupt handler whenever the interrupt is enabled.

An illustrative example of this mechanism is depicted in Figure 4.12 where a task t is analyzed with two preempting interrupts. Let us denote by X the abstract environment reaching the statement s : `x = e`. The approximation of the eventual preemption before s is performed by merging X with the return contexts of the enabled interrupts before passing the resulting environment to the transfer function of the statement. In addition, if X contains new state information not present in the current preemption contexts, the latter should be updated in order to perform a new iteration that will compute the new return contexts that consider these modifications.

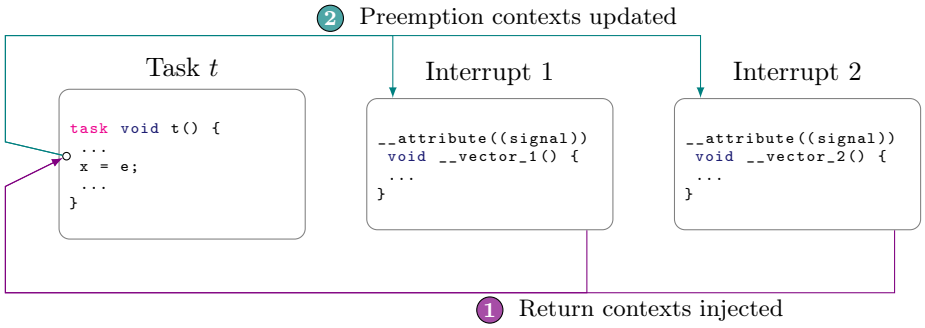


Figure 4.12: Approximation of interrupts preemption using the return contexts of interrupts. Preemption contexts are updated for an eventual next iteration.

$$
\begin{array}{l}
\wr \textit{ Loop until the preemption contexts stabilize } \int \\
\mathsf{lfp}\ \lambda(X_p, X_r). \\
\quad \wr \textit{ Main program is analyzed first } \int \\
\quad \mathsf{let}(-, X_p^{main}, X_r^{main}) = \\
\quad\quad \wr \textit{ Initial abstract state } \int \\
\quad\quad \mathsf{let}\ X_0 = \bot_{\mathcal{Q}}[\varnothing_{\mathcal{Q}} \to \bot_{\mathcal{S}}[s_0 \to \mathsf{fold}\ \ (\lambda r.\ \lambda X.\ \mathbf{S}[\![r = 0]\!]^\sharp_{\mathcal{M}} X)\ \top_{\mathcal{M}} \mathcal{R}]]\ \ \mathsf{in} \\
\quad\quad \wr \textit{ Analyze the initialization of the kernel and the drivers } \int \\
\quad\quad \mathsf{let}\ X_1 = \mathbf{S}[\![I_{drv}]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![I_{ker}]\!]^\sharp_{\mathcal{I}}(X_0, X_p, X_r)\ \ \mathsf{in} \\
\quad\quad \wr \textit{ Analyze tasks } \int \\
\quad\quad \mathsf{let}\ X_2 = \mathsf{lfp}\ \lambda X.\ X \triangledown_{\mathcal{I}} (X_1 \sqcup_{\mathcal{I}} \bigsqcup_{t \in \mathbb{T}}^{\mathcal{I}} \mathbf{S}[\![body(t)]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![\mathsf{dequeue}\ t]\!]^\sharp_{\mathcal{I}} X)\ \ \mathsf{in} \\
\quad\quad \wr \textit{ Analyze user apps initialization } \int \\
\quad\quad \mathsf{let}\ X_3 = \mathbf{S}[\![\texttt{oldSREG = SREG}]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![I_{app}]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![\texttt{sei}]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![\mathsf{notask}]\!]^\sharp_{\mathcal{I}} X_2\ \ \mathsf{in} \\
\quad\quad \mathsf{lfp}\ \lambda X.\ X \triangledown_{\mathcal{I}} (\\
\quad\quad\quad \wr \textit{ Restore the global interrupts mask } \int \\
\quad\quad\quad \mathsf{let}\ X_4 = \mathbf{S}[\![\texttt{SREG = oldSREG}]\!]^\sharp_{\mathcal{I}} X\ \ \mathsf{in} \\
\quad\quad\quad \wr \textit{ Analyze tasks } \int \\
\quad\quad\quad \mathsf{let}\ X_5 = \mathsf{lfp}\ \lambda X'.\ X' \triangledown_{\mathcal{I}} (X_4 \sqcup_{\mathcal{I}} \bigsqcup_{t \in \mathbb{T}}^{\mathcal{I}} \mathbf{S}[\![body(t)]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![\mathsf{dequeue}\ t]\!]^\sharp_{\mathcal{I}} X')\ \ \mathsf{in} \\
\quad\quad\quad \wr \textit{ Move the MCU to sleep mode } \int \\
\quad\quad\quad \mathsf{let}\ X_6 = \mathbf{S}[\![\mathsf{event}^\star\ \texttt{sleep}]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![\texttt{sei}]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![\texttt{oldSREG = SREG}]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![\mathsf{notask}]\!]^\sharp_{\mathcal{I}} X_5\ \ \mathsf{in} \\
\quad\quad\quad \wr \textit{ Interrupts are not part of the main programs } \int \\
\quad\quad\quad X_3 \sqcup_{\mathcal{I}} X_6 \\
\quad\quad) \\
\quad \mathsf{in} \\
\quad \wr \textit{ Interrupts are analyzed separately } \int \\
\quad \mathsf{let}\ X^{int} = \lambda i.\ \mathbf{S}[\![\texttt{sei}]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![body(i)]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![\texttt{cli}]\!]^\sharp_{\mathcal{I}} \circ \mathbf{S}[\![\mathsf{event}^\star\ \texttt{int}_i]\!]^\sharp_{\mathcal{I}}(X_p(i), X_p, X_r)\ \ \mathsf{in} \\
\quad \wr \textit{ Extraction of the new return contexts } \int \\
\quad \mathsf{let}\ X'_r = \lambda i.\ (\mathsf{let}(X_i, -, -) = X^{int}(i)\ \ \mathsf{in}\ X_i)\ \ \mathsf{in} \\
\quad \wr \textit{ Extraction of the new preemption contexts } \int \\
\quad \mathsf{let}\ X'_p = \lambda i.\ X_p(i) \triangledown (X_p^{main}(i) \sqcup_{\mathcal{K}} \bigsqcup_{j \neq i \in \mathbb{I}}^{\mathcal{K}} \mathsf{let}(-, X_p^j, -) = X^{int}(j)\ \ \mathsf{in}\ X_p^j(i))\ \ \mathsf{in} \\
\quad (X'_p, X'_r)
\end{array}
$$

Figure 4.13: Abstract interpreter for preemptive executions.

Formally, we define our preemptive abstract domain $\mathcal{D}^\sharp_\mathcal{I}$ as the following product:

$$\mathcal{D}^\sharp_\mathcal{I} \triangleq \mathcal{D}^\sharp_\mathcal{K} \times (\mathbb{I} \to \mathcal{D}^\sharp_\mathcal{K}) \times (\mathbb{I} \to \mathcal{D}^\sharp_\mathcal{K})$$

The first element of this product corresponds to the abstract environment of the current flow over which statements are executed. The second and the third elements represent two maps giving for every interrupt its preemption and return contexts respectively. The definitions of $\bot_\mathcal{I}$, $\sqsubseteq_\mathcal{I}$, $\sqcup_\mathcal{I}$ and $\triangledown_\mathcal{I}$ are easily derived from this definition. For an atomic statement s, we define its abstract transfer function as:

$$\begin{array}{l}
\mathbf{S}[\![s]\!]^\sharp_\mathcal{I}(X_c, X_p, X_r) \triangleq \\
\quad \textsf{let } I_{en} = \{i \in \mathbb{I} \mid \mathbf{S}[\![?\,\mathsf{icond}(i)]\!]^\sharp_\mathcal{K} \circ \mathbf{S}[\![?\,\mathsf{gcond}]\!]^\sharp_\mathcal{K} X_c \neq \bot_\mathcal{K}\} \textsf{ in} \\
\quad \textsf{let } X'_c = \mathbf{S}[\![?!\,\mathsf{gcond}]\!]^\sharp_\mathcal{K} X_c \sqcup_\mathcal{K} \bigsqcup_{\mathcal{K}, i \in I_{en}} X_r(i) \textsf{ in} \\
\quad \textsf{let } X'_p = \lambda i. \begin{cases} X_c \sqcup_\mathcal{K} X_p(i) & \text{if } i \in I_{en} \\ X_p(i) & \text{otherwise} \end{cases} \textsf{ in} \\
\quad (\mathbf{S}[\![s]\!]^\sharp_\mathcal{K} X'_c, X'_p, X_r)
\end{array}$$

The intuition behind this definition can be explained as follows. First, we construct the set I_{en} of enabled interrupts using the enable mask expressions. After that, we merge their return contexts with the current abstract environment X_c to over-approximate the effect of the nondeterministic preemption. Also, we update the preemption contexts of the enabled interrupts with X_c. Finally, we apply the statement abstract transfer function $\mathbf{S}[\![s]\!]^\sharp_\mathcal{K}$ on the newly computed environment X'_c.

Our modular abstract interpreter for the analysis of preemptive executions is presented in Figure 4.13 and operates as follows. Given the input preemption and return contexts X_p and X_r, we execute the *main* TinyOS program, which consists in executing the different initialization procedures and then entering the infinite tasks-sleep-interrupt loop, but without executing the interrupt handlers. During the analysis of the main program, the functions $\mathbf{S}[\![.]\!]^\sharp_\mathcal{I}$ collect the preemption contexts of every interrupt. After reaching the fixpoint of the infinite loop, we execute every interrupt on its preemption context. As for the main program, we also collect during the analysis the preemption contexts of every interrupt. To compute the new preemptive and return contexts $X'_p(i), X'_r(i)$ of an interrupt i for the next iteration, we proceed as follows. For $X'_r(i)$, we just retrieve the post-execution environment reached at the end of the analysis of the vector of interrupt i. For the preemption context $X'_p(i)$, we merge the environments $X^{main}_p(i)$ collected during the analysis of the main program with those collected during the analysis of other interrupts. Note that this formulation assumes that there is no reentrant interrupt, i.e. an interrupt does not allow, during its execution, being interrupted by itself. Finally, to accelerate the convergence of the fixpoint

computation, we use the widening operator when computing the new preemption contexts $X'_p(i)$.

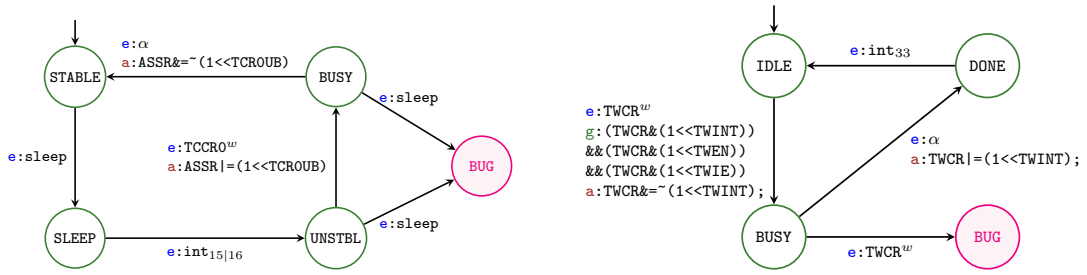
4.6 Experiments

In this section we describe the experimental results of the analysis of our motivating example and other real-world TinyOS device drivers using a prototype of our analysis called SADA (*Static Analyzer with Device Abstraction*) that supports both sequential and preemptive execution models. We implemented SADA using the OCaml language. The implementation consists of 4.000 lines of code and uses the CIL framework ([NMRW02]) for parsing the input C files generated by the `ncc` compiler. It also builds upon the Apron library developed by [JM09] that provides a rich collection of numerical abstract domains, such as intervals, octagons and polyhedra. For our experiments, we used the interval domain enriched with modular arithmetics operations to handle the finite-size representation of numbers.

To assess the efficiency and precision of SADA, we first analyzed some device drivers of the ATmega128 MCU from the latest TinyOS 1.x release. We chose three test cases with growing complexities, in terms of lines of codes and the tasks/interrupts execution flows. For each case, a set of ADPs were verified and we were interested in three metrics: the analysis time, the peak memory consumption and the nature of the reported alarms. In total, seven ADPs were analyzed that capture the most recurrent programming patterns in embedded device driver development.

The second set of experiments consists in the analysis of the same ADPs but on a different implementation, that is, on the version 2.x of TinyOS. It is worth noting that TinyOS 2.x has been completely re-written with drastic changes in the design and the implementation. Therefore, analyzing different versions of the same driver, while keeping the ADPs unchanged, allows showing the capacity of the tool in analyzing the same specifications but on several, and possibly extremely different, implementations without additional effort from the user to configure the tool or accommodate the source code.

Finally, we also run these experiments using i-CBMC in order to compare its performances to SADA. The reason behind this choice is that it is the most efficient state-of-the-art analysis tool available for interrupt-based programs ([KLM+15]). However, we limited the use of i-CBMC to the analysis of TinyOS 2.x device drivers only because using i-CBMC requires a considerable amount of time in instrumenting the source for emulating the asynchronous hardware operations and the arrival of interrupts, as explained later in this section.



(a) Stabilization of Timer/Counter0. (b) TWI serial transfer using interrupts.

Figure 4.14: Examples of ADPs used in the experiments.

4.6.1 Test Cases

Embedded device driver development shares many programming practices that can be applied to different hardware architectures. Polling on status bits, interrupt-based serial transfer and GPIO configurations are some examples of frequent patterns that represent important building blocks in most implementations of device drivers. In this section, we briefly describe some instantiations of these recurrent patterns on the ATmega128 platform, along with some functional properties for ensuring their correctness.

Asynchronous Timer

The ATmgega128 provides four hardware timers with different capabilities and applications. The 8-bit Timer/Counter0 is frequently used in low-power embedded applications since it is the only timer that allows going to a deep sleep mode (in terms of energy consumption) while keeping the timer module active to wake up the MCU after a period of time. To do so, Timer/Counter0 should be configured in *asynchronous mode* that allows it to use an external 32.768kHz crystal (`TOSC1`) to operate independently from the main oscillator of the MCU.

However, the asynchronous mode of Timer/Counter0 requires a number of safety measures as listed in the datasheet of the MCU ([Atm11, pp. 106–108]). We limit the description herein to two important ones:

- The first precaution that should be considered when operating in asynchronous mode is the stabilization of the timer after wakeup. Indeed, the datasheet stipulates that "*if the time between wakeup and re-entering sleep mode is less than one `TOSC1` cycle, the interrupt will not occur, and the device will fail to wake up*".
- Also, the datasheet indicates that "*when writing to one of the [timer] registers, the value is transferred to a temporary register, and latched after two positive edges on `TOSC1`. The user should not write a new value before the contents of the*

Temporary Register have been transferred to its destination".

To ensure both requirements, the same mechanism is generally employed, which is based on polling the first three bits of the `ASSR` register that indicate the effective transfer from the temporary register to the actual register. Since this operation requires at least one `TOSC1` cycle, the timer driver can assess the value of these bits for ensuring both stabilization and correct transfer to registers.

Consequently, we wrote three ADPs to ensure that the driver performs the appropriate polling mechanism. The first one, denoted $\mathcal{A}_{\texttt{STBL}}$, specifies the stabilization requirement and verifies that, when the MCU is waked-up by the timer routine, at least one of the timer registers (`OCR0`, or `TCCR0`) is modified and the program will not return to sleep again only until it verifies that appropriate status bit in `ASSR` indicates the end of transfer. An illustration of this ADP is depicted in Figure 4.14(a). The two other ADPs, denoted $\mathcal{A}_{\texttt{OCR0}}$ and $\mathcal{A}_{\texttt{TCCR0}}$, model the proper access to registers `OCR0` and `TCCR0` respectively, and ensure that the driver waits after every write access for the completion of the transfer operation before modifying the register again.

ADG715 Analog Switch

The component ADG715 is an analog switch that is used generally as a multiplexer to dynamically route the power supply to other components (mainly sensors) for controlling energy consumption. It is configured by the MCU through a TWI (*Two Wire Interface*) serial bus for sending byte commands to open/close the switch ports.

To ensure the proper transfer of these commands over the bus, several safety rules should be observed. In our experiments, we were interested in two properties:

- It is important to ensure that all TWI-related hardware registers are not be modified when the bus is busy. In the case of TinyOS (both 1.x and 2.x), TWI operations are performed in the interrupt-based mode. Therefore, the $\mathcal{A}_{\texttt{TWI-TX}}$ ADP, depicted in Figure 4.14(b), tracks the start of a transmission and performs an asynchronous transition to model the arrival of the interrupt at any moment. All access to the register in the meanwhile are forbidden.

- In addition to safe serial transfer, the `SCL` and `SDL` pins, used as the clock and data lines respectively, should be configured as pulled-up. Our ADP $\mathcal{A}_{\texttt{PULL-UP}}$ verifies this condition by checking that the corresponding GPIO pins are configured as input pins through the `DDx` register and that they are driven high through the `PORTx` register. This check is performed at every transmission over the TWI bus.

CC2420 Transceiver

In Section 4.2, we described in detail the ADP $\mathcal{A}_{\texttt{SPI-TX}}$ that models a safe serial transfer between the MCU and an SPI slave (which is the CC2420 transceiver in the case of the MicaZ mote). In addition, our benchmark includes a second ADP $\mathcal{A}_{\texttt{SPI-SS}}$ that ensures that the MCU selects the appropriate end-point slave by pulling down a particular `SS` (*Slave Select*) pin before starting transmitting over the SPI bus. Consequently, the ADP checks that direction and state of PB0 pin are correctly set in registers `DDRB` and `PORTB` whenever a byte is written into the data register `SPDR`.

4.6.2 TinyOS Benchmarks

Table 4.1 shows the results of our tool SADA on the different TinyOS device drivers. In order to show the cost of introducing arbitrary preemption during the analysis, we considered both sequential and preemptive execution models separately. In addition, we compared the results of the hardware state partitioning domain $\mathcal{D}_{\mathcal{S}}$ with the tasks queue partitioning domain $\mathcal{D}_{\mathcal{Q}}$. Overall, our benchmarks consisted of 112 tests and we analyzed a total of 23260 lines of code, with drivers containing between 1 to 10 tasks and 1 or 2 interrupts.

The analysis terminated before timeout in 95% of the test cases, with 90% of them under one minute. We experienced only 6 timeouts. Four of them were due to the coarse over-approximation of the state partitioning domain $\mathcal{D}_{\mathcal{S}}$ that does not keep information about the tasks queue, but these timeouts have been eliminated by the use of the more precise domain $\mathcal{D}_{\mathcal{Q}}$. The two remaining ones emerged when analyzing the incorrect versions of the drivers. Nevertheless, it is important to note that these benchmarks were run in full-coverage mode of SADA, *i.e.* the analysis does not stop until all paths are verified. SADA supports also another option that terminates the analysis whenever an error is detected. In such configuration, all timeouts disappear, as described later in this section (see Table 4.2).

In terms of precision, we note that SADA has not missed any bug on the incorrect version of the drivers, which is coherent with the soundness property of the underlying abstract interpretation theory. In addition, two real bugs were detected on the original versions, which were, to our knowledge, not known before. On the other hand, 7 false alarms were detected. Using the tasks queue partitioning domain $\mathcal{D}_{\mathcal{Q}}$ we can eliminate 4 of them, in a similar way to the example in Figure 4.9 that motivated the introduction of $\mathcal{D}_{\mathcal{Q}}$. The remaining 3 false alarms are all related to the timer driver and are due to the lack of quantitative modeling of the physical time and delays. Indeed, asynchronous events – such as propagating a value to the `TCCR0` from the temporary register – are

Table 4.1: Analysis benchmarks for TinyOS 1.x and 2.x. Three metrics are shown: the analysis time (in seconds), the peak memory consumption (in mega bytes) and the analysis result (✔: safe, ✘: bug detected correctly, (✘): false bug alarm, (✔): bug missed). ∞ denotes a timeout of 30mn.

						Non-Preemptive Analysis								Preemptive Analysis							
						$\mathcal{D}_{\mathcal{S}}$				$\mathcal{D}_{\mathcal{Q}}$				$\mathcal{D}_{\mathcal{S}}$				$\mathcal{D}_{\mathcal{Q}}$			
Driver	# LoC	# ISR	$\lvert\mathbb{T}\rvert$	ADP	$\lvert\mathcal{S}\rvert$	Original			Incorrect			Original			Incorrect			Original			Incorrect		
Timer 1.x	1627	1	3	$\mathcal{A}_{\texttt{STBL}}$	7	1	10	(✘)	1	10	✘	1	10	✔	1	10	✘	6	11	(✘)	7	11	✘	39	16	(✘)	36	16	✘
				$\mathcal{A}_{\texttt{OCRO}}$	4	1	10	(✘)	1	10	✘	1	10	✔	2	10	✘	3	10	(✘)	3	10	✘	29	15	(✘)	29	15	✘
				$\mathcal{A}_{\texttt{TCCRO}}$	4	1	10	✔	1	10	✘	1	10	✔	1	10	✘	3	10	✔	3	10	✘	37	16	✔	41	15	✘
Timer 2.x	2384	2	2	$\mathcal{A}_{\texttt{STBL}}$	7	2	11	✔	3	11	✘	2	11	✔	3	11	✘	7	12	✔	15	12	✘	26	13	✔	78	15	✘
				$\mathcal{A}_{\texttt{OCRO}}$	4	3	11	✔	2	11	✘	4	11	✔	3	11	✘	10	12	✔	9	12	✘	38	13	✔	36	13	✘
				$\mathcal{A}_{\texttt{TCCRO}}$	4	3	11	✔	3	11	✘	3	11	✔	3	11	✘	10	11	(✘)	10	12	✘	23	13	(✘)	23	13	✘
ADG715 1.x	2038	1	1	$\mathcal{A}_{\texttt{PULL-UP}}$	4	1	11	✘	1	11	✘	1	11	✘	1	10	✘	1	11	✘	1	11	✘	1	11	✘	1	11	✘
				$\mathcal{A}_{\texttt{TWI-TX}}$	6	1	11	✔	1	11	✘	1	11	✔	2	11	✘	4	11	✔	4	11	✘	7	12	✔	7	12	✘
ADG715 2.x	4412	1	6	$\mathcal{A}_{\texttt{PULL-UP}}$	4	3	13	✔	3	13	✘	2	13	✔	2	13	✘	23	14	✔	30	14	✘	8	13	✔	6	14	✘
				$\mathcal{A}_{\texttt{TWI-TX}}$	6	4	14	✘	6	14	✘	2	14	✘	3	14	✘	40	16	✘	42	16	✘	6	14	✘	6	14	✘
CC2420 1.x	2666	1	1	$\mathcal{A}_{\texttt{SPI-SS}}$	4	10	13	✔	4	13	✘	28	12	✔	2	12	✘	12	12	✔	6	12	✘	850	42	✔	52	11	✘
				$\mathcal{A}_{\texttt{SPI-TX}}$	10	5	12	✔	3	12	✘	28	13	✔	26	12	✘	21	13	✔	19	14	✘	1600	74	✔	∞		
CC2420 2.x	10133	2	10	$\mathcal{A}_{\texttt{SPI-SS}}$	4	1040	33	(✘)	800	33	✘	12	17	✔	5	16	✘	∞			∞			34	18	✔	27	18	✘
				$\mathcal{A}_{\texttt{SPI-TX}}$	10	1659	27	(✘)	597	27	✘	13	17	✔	∞			∞			∞			39	18	✔	49	19	✘

Table 4.2: Comparison with i-CBMC on the TinyOS 2.x drivers with different unwinding iteration limits.

	Non-Preemptive Analysis												Preemptive Analysis											
	i-CBMC (1 iteration)				i-CBMC (2 iterations)				SADA				i-CBMC (1 iteration)				i-CBMC (2 iterations)				SADA			
ADP	Original		Incorrect		Original		Incorrect		Original		Incorrect		Original		Incorrect		Original		Incorrect		Original		Incorrect	
$\mathcal{A}_{\texttt{STBL}}$	38	200 ✔	29	200 (✔)	456	447 ✔	354	441 ✘	2	11 ✔	1	9 ✘	45	290 ✔	43	286 (✔)	404	494 (✘)	386	497 ✘	26	13 ✔	6	12 ✘
$\mathcal{A}_{\texttt{OCRO}}$	16	178 ✔	35	247 (✔)	325	450 ✔	401	474 ✘	4	11 ✔	1	8 ✘	33	262 ✔	64	340 (✔)	354	500 ✔	444	561 ✘	38	13 ✔	1	9 ✘
$\mathcal{A}_{\texttt{TCCRO}}$	29	210 ✔	34	225 (✔)	357	462 ✔	400	484 ✘	3	11 ✔	1	10 ✘	54	323 (✘)	56	336 ✘	385	510 (✘)	397	507 ✘	4	10 (✘)	3	10 ✘
$\mathcal{A}_{\texttt{PULL-UP}}$	2	37 ✔	1	37 (✔)	1601	1919 ✔	1552	1907 (✔)	2	13 ✔	1	11 ✘	1	37 ✔	1	41 ✘	1	36 ✔	1	37 (✔)	8	13 ✔	4	12 ✘
$\mathcal{A}_{\texttt{TWI-TX}}$	1	37 (✔)	1	36 (✔)	1362	1762 (✔)	1520	1940 (✔)	1	10 ✘	1	11 ✘	1	36 (✔)	1	37 (✔)	1	36 (✔)	1	37 (✔)	1	11 ✘	1	11 ✘
$\mathcal{A}_{\texttt{SPI-SS}}$	2	64 ✔	1	62 (✔)	129	507 ✔	126	505 (✔)	12	17 ✔	4	16 ✘		∞		∞		∞		∞	34	18 ✔	6	16 ✘
$\mathcal{A}_{\texttt{SPI-TX}}$	2	65 ✔	2	62 (✔)		∞		∞	13	17 ✔	1	14 ✘		∞		∞		∞		∞	39	18 ✔	2	15 ✘

modeled in our analysis using nondeterminism and can occur at any moment. However, these transitions in fact have a limited trigger timeframe, after which we are sure that the asynchronous event has occurred. This type of information is not handled by SADA and is out of the scope of the ADPs semantics. Note that other existing analysis tools such as i-CBMC do not model such semantics and are therefore prone to the same problem.

From these results, we can observe that the analysis costs do not always increase with the level of precision of the abstract domain. Indeed, in most cases, the queue partitioning domain $\mathcal{D}_{\mathcal{Q}}$ has shown a better analysis time compared to the more compact state abstraction $\mathcal{D}_{\mathcal{S}}$. This is particularly observable when the driver implementation uses a significant number of tasks, such as the CC2420 2.x driver where we obtained a 95% decrease in the analysis time. This is explained by the fact that, due to the additional details provided by the increased precision of the domain, more spurious execution paths are filtered and fewer iterations are required to reach the fixpoint. However, for less task-intensive programs such the Timer driver, the domain $\mathcal{D}_{\mathcal{Q}}$ was less efficient.

Finally, these experimental results demonstrate the scalability of the Modular Abstract Interpretation framework, since the sound preemptive analysis was able to analyze the full search space with arbitrary preemption, while maintaining a reasonable cost in comparison to the restricted sequential analysis which does not necessarily cover all behaviors.

4.6.3 Comparison with i-CBMC

Table 4.2 shows the obtained results of running the TinyOS 2.x test cases using i-CBMC with different loops unwinding. The table also presents the results of SADA without full-coverage in case of error detection, since i-CBMC behaves in the same way. To compare both approaches, we consider three criteria: efficiency, precision and automation.

Efficiency. It is clear that SADA scales better than i-CBMC in all test cases. Analysis times of i-CBMC are larger in general with a total of 10 timeouts, while SADA converged in all cases without exceeding one minute per driver. Note that we limited the maximal unwinding of the main TinyOS loop to two iterations in the experiments with i-CBMC, since the analysis time of i-CBMC exceeded the timeout duration when using three loops unwinding for all the benchmark programs. In addition, the memory consumption of i-CBMC is much higher, reaching the giga byte in many cases, in contrast to SADA that consumed at most 20 mega bytes in the worst case thanks to the

use of the efficient abstraction of the interval domain.

Precision. Due to the limited search depth in i-CBMC, not all errors can be discovered. This is exemplified by the missed bugs reported during the analysis of the incorrect versions of the drivers. SADA, and abstract interpretation based tools in general, do not suffer from these limitations since all possible errors are detected, which makes the tool more adequate to correctness certification. False alarms are, on the other hand, the main drawback of our method since no indication can be provided to developers to decide the genuineness of the errors. However, in practice, SADA presents a low false alarm rate with only one occurrence in TinyOS 2.x benchmark. Note that i-CBMC reported also the same false alarm because both tools lack appropriate modeling of hardware-level timings in order to restrict the firing timeframes of the critical interrupts.

Automation. As reported by [BK11], employing a bounded model checking approach is generally hampered by the laborious task of setting the appropriate loops unwinding which requires enumerating all loops of the program, eliminating the unnecessary ones and fixing an individual unwinding limit for the remaining loops (an exception was made for the main TinyOS loop for which we made its unwinding limit as a parameter of the analysis to vary the depth of execution flows). In the case of SADA, such manual tuning was not required thanks to the widening mechanism of abstract interpretation that allows a fully automatic and sound analysis up to arbitrary (possibly infinite) loop bounds. Additionally, i-CBMC is a general purpose tool for analyzing preemptive ANSI C programs and does not embed a dedicated semantics for low-level hardware interactions. As a consequence, it is necessary to manually modify the programs in order to emulate the reaction of the device to register modification. Practically, before analyzing a program with i-CBMC, we added C functions modeling the behavior of hardware in reaction to read/write register events and we inserted asynchronous calls to these functions and to interrupt vectors at the appropriate locations. Note that we were not able to faithfully mimic the full semantics of the driver due to some limitations in i-CBMC related to the management of atomic sections. Consequently, the obtained instrumented program is not semantically equivalent to the real behavior of the driver and the device.

4.7 Conclusion

We presented an effective static analysis by Abstract Interpretation of device drivers in TinyOS programs. We described an automata-based formalism to express functional

properties that specify the correct hardware interaction patterns that should be followed by the device driver. Our analysis is based on several abstract domains that provide a multi-level partitioning according to the hardware state, the tasks queue and the interrupts masking system. To efficiently handle concurrency, we perform a compositional analysis by processing the interrupt vectors separately and propagate their effects to program locations where interrupts are enabled. Several experiments were conducted on real-world TinyOS drivers and promising results demonstrate the efficiency of the approach.

Chapter 5

Quantitative Verification of Communication Protocols

Wireless nodes are characterized by their limited capabilities in terms of energy, memory and bandwidth. A reliable program has to verify the functional safety properties to avoid bugs, and should additionally provide efficient resources management policies for optimizing its consumption. This efficiency is determined by evaluating a set of performance indicators and check that they fit within the constraints/requirements of the application. It is fundamental to notice that these metrics have an intrinsic *probabilistic* semantics induced by many random aspects of wireless embedded networks, such as the wireless link quality for example. In this chapter, we illustrate an application of the theory of abstract interpretation for evaluating the performance of an embedded wireless program by taking into account the stochastic nature of the behaviors.

5.1 Introduction

The analysis of probabilistic programs represents a challenging problem. The difficulty comes from the fact that execution traces are characterized by probability distributions that may have very complex forms since they are affected by dynamic behavior of program. In addition, in such stochastic context, programmers and system designers are interested in quantitative properties not allowed by conventional semantics analysis, such as the expectation of system performance metrics (like energy consumption, computation time, etc.) or the probability of reaching bug states.

Proportional Expectations. Communication protocols are the most recurrent instance of probabilistic programs in the context of wireless embedded networks (and the only one in many software stacks). When analyzing the performance of a communication protocol with respect to a given indicator (such as the energy for example), we generally want to compute the average change of this indicator *in one time unit*, instead of computing its *absolute* expectation. Indeed, these *proportional expectations* are fundamental in performance evaluation of communication protocols, and include for example packet throughput (number of packets successfully transmitted per second) or the transceiver duty cycle (proportion of time the radio is activated). The importance of such metrics comes from the fact that protocols run generally as an infinite loop without a limited time-frame. Consequently, computing the expectation of absolute quantities is not useful since they keep accumulated along executions and are therefore unbounded as executions are infinite. However, their *proportional expectation* gives the velocity of the performance indicator, which is clearly a more meaningful (and hopefully bounded) quantitative representation that can be used to compare between different resource usage policies.

Motivation. We are interested in quantifying these metrics (i) *automatically* by analyzing the source code, (ii) *soundly* by considering all executions and (iii) *symbolically* by expressing the result in terms of the protocol parameters. To our knowledge, no existing approach can perform analysis for possibly infinite state systems. Indeed, many proposed solutions [SCG13, CS13] focus on computing the probability of some program assertions which is not helpful in deriving proportional expectations. On the other hand, solutions for computing (classical) expectations [CS14, BEFFH16] for infinite systems can not be extended easily to proportional expectations. In our understanding, these solutions are limited by the fact that they can support only linear expressions, while a proportional expectation is by definition a ratio between a performance indicator variable and a time variable. Finally, parametric Markov chains analyzers such as PARAM [HHZ11] can compute proportional expectations, but are limited to finite state systems.

Example 5.1. To illustrate our motivation, we consider a simple communication scheme depicted in Figure 5.1(a). It implements an Unbounded Retransmission with Backoff (URB) mechanism that works as follows. The sender begins by acquiring some readings from its sensing device using the function `sense`. To ensure reliable communication of this data over the lossy wireless link, the packet is successively transmitted until receiving an acknowledgment from the destination, which is done via the function `unicast`. In addition, to avoid collisions, a random backoff is performed using a uniform distribution on the range $[1, B]$, where B is a parameter of the protocol.

Finally, to reduce the energy usage, the sender enters sleep mode after every successful transmission and before acquiring the new readings. The duration of the sleep period is determined by a parameter S.

Let us assume that transmissions obey a Bernoulli distribution with parameter p. We are interested in measuring the average value of the throughput θ as a function of the parameters B and S. To do so, we model the protocol using a discrete time Markov chain, as illustrated in Figure 5.1(b). This modeling allows us to derive the throughput θ by computing the stationary distribution $\boldsymbol{\pi}$ of the chain, which characterizes the long-run time proportion spent in every state of the chain. Clearly, the proportional expectation θ is equal to $\boldsymbol{\pi}_{tx}$ since the state tx is visited *iff* a packet is successfully transmitted.

Existing verification tools can not obtain this information symbolically (i.e. as a function of the parameters), and the main reason is that the chain illustrated in Figure 5.1(b), besides being possibly unbounded, is not actually a standard Markov chain. Indeed, since the number of states (and therefore the structure of the chain itself) depends on the parameter B, we have in fact a *family* of Markov chains; that is, one different chain for every value of B. Existing tools, such as PRISM with its extension PARAM, do not support these structures. ○

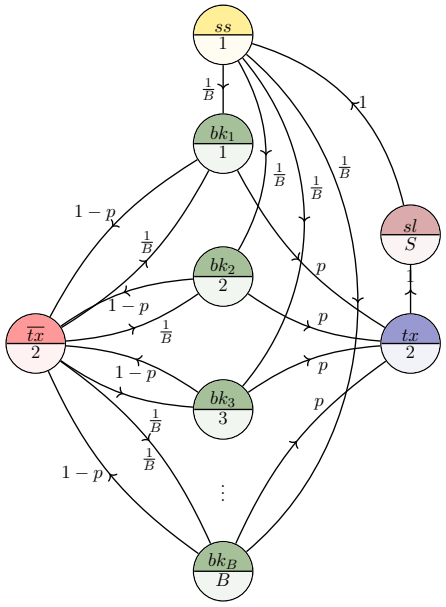
Contributions. To overcome this problem, we propose a static analysis technique based on three main contributions:

- First, we introduce a novel notion of *Abstract Markov Chains* that can over-approximate a family of possibly unbounded discrete time Markov chains. These abstract chains are inferred automatically by analyzing the source code of the program, are guaranteed to have a finite size while covering all possible probabilistic traces of the program.
- Our second contribution is a *soundness result* that allows us to derive from the abstract Markov chain a system of parametric linear inequalities for bounding the concrete stationary distribution.
- Finally, our last contribution is a *method for solving systems of parametric linear inequalities* using a symbolic Fourier-Motzkin elimination. By applying it on the system obtained through the soundness result, we can derive symbolic and guaranteed bounds of the proportional expectation of interest.

Example 5.2. To give an idea about the results of our analysis, we show in Figure 5.2(a) the obtained abstract Markov chain of the URB example. We notice that the set of states $\{b_1, b_2, b_3, \ldots, b_B\}$ has been approximated with two abstract states bk_1 and

```
1  while(1) {
2    data = sense();
3    //Trasmission loop
4    while(1) {
5      //Uniform backoff
6      sleep(uniform(1, B));
7      if (unicast(data))
8        break;
9    }
10   sleep(S);
11 }
```

(a)



(b)

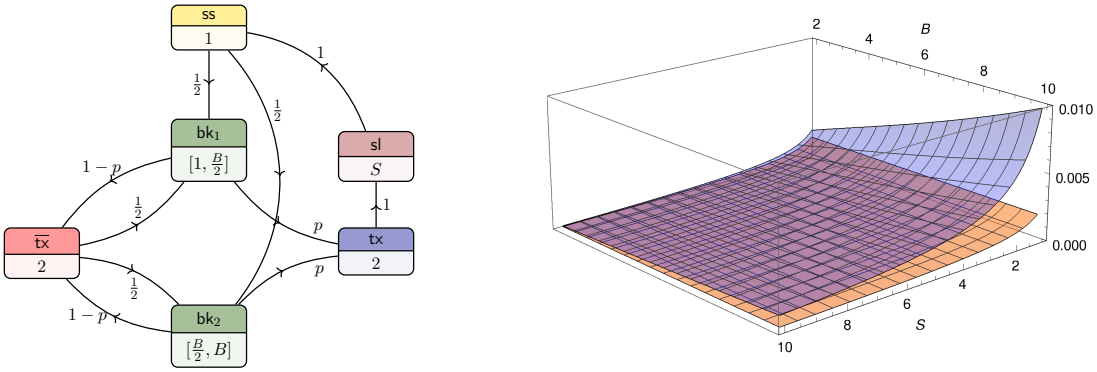
Figure 5.1: Motivating example. (a) Unbounded Retransmission with Backoff (URB) protocol. (b) The associated discrete time Markov chain. The lower part of each state indicates the number of time units spent in the state, by assuming that the `sense()` function consumes one tick, and `unicast()` consumes two ticks for sending and receiving the acknowledgment.

bk_2. After constructing the abstract chain, we apply the soundness result to infer the following system of inequalities that bounds the concrete stationary distribution:

$$\begin{cases} \boldsymbol{\pi}_{\mathsf{ss}} \leq \frac{1}{S}\boldsymbol{\pi}_{\mathsf{sl}} \\ \boldsymbol{\pi}_{\mathsf{bk}_1} \leq \frac{B}{4}\boldsymbol{\pi}_{\mathsf{ss}} + \frac{B}{8}\boldsymbol{\pi}_{\overline{\mathsf{tx}}} \\ \boldsymbol{\pi}_{\mathsf{bk}_2} \leq \frac{B}{2}\boldsymbol{\pi}_{\mathsf{ss}} + \frac{B}{4}\boldsymbol{\pi}_{\overline{\mathsf{tx}}} \\ \boldsymbol{\pi}_{\mathsf{tx}} \leq 2p\boldsymbol{\pi}_{\mathsf{bk}_1} + \frac{4p}{B}\boldsymbol{\pi}_{\mathsf{bk}_2} \\ \boldsymbol{\pi}_{\overline{\mathsf{tx}}} \leq 2(1-p)\boldsymbol{\pi}_{\mathsf{bk}_1} + \frac{4(1-p)}{B}\boldsymbol{\pi}_{\mathsf{bk}_2} \\ \boldsymbol{\pi}_{\mathsf{sl}} \leq \frac{S}{2}\boldsymbol{\pi}_{\mathsf{tx}} \\ \boldsymbol{\pi}_{\mathsf{ss}} + \boldsymbol{\pi}_{\mathsf{bk}_1} + \boldsymbol{\pi}_{\mathsf{bk}_2} + \boldsymbol{\pi}_{\mathsf{tx}} + \boldsymbol{\pi}_{\overline{\mathsf{tx}}} + \boldsymbol{\pi}_{\mathsf{sl}} = 1 \end{cases} \tag{5.1}$$

Since we are searching for the bounds of $\boldsymbol{\pi}_{\mathsf{tx}}$, we apply our parametric resolution algorithm to eliminate all other unknowns. The obtained symbolic bounds are shown in Figure 5.2(b) as a function of S and B with $p = 0.9$. The less is the gap between the lower and upper bounds, the more precise is the analysis. ○

Limitations. Our approach is still in a preliminary development phase and presents some limitations. First, the analysis is limited to discrete probability distributions,



(a) (b)

Figure 5.2: Analysis results of the unbounded retransmissions protocol with uniform backoff. (a) Abstract Markov chain. (b) Variation of the upper and lower bounds of throughput θ w.r.t. the parameters S and B while fixing $p = 0.9$.

such as Bernoulli and Uniform distributions over integers. Secondly, we have not yet full support of TinyOS semantics, so we limit the description herein to a simple C-like language. Finally, we do not consider pure non-deterministic statements, such as `x = [1, 5];`. Instead, we assume that such non-determinism is resolved by means of probability distributions, like `x = uniform(1, 5);`.

Outline. The remaining of the chapter is organized as follows. We present in Section 5.2 the concrete semantics of our probabilistic programs. After that, Section 5.3 introduces our Abstract Markov Chain domain and presents its abstract semantics. In Section 5.4, we explain the soundness results for extracting the system of parametric linear inequalities and we show in Section 5.5 how to solve it. The results of the preliminary experiments are presented in Section 5.6. Finally, Section 5.7 concludes the chapter.

5.2 Concrete Semantics

We consider communication protocols that can be represented as (possibly infinite) discrete time Markov chains, as it is one of the mostly widespread stochastic models for performance evaluation used by networking community. For describing these protocols, we use a simplified probabilistic language having a C-like syntax based on our toy language presented in Section 3.2.1. In addition to the classical statements of assignments, `if` conditionals and `while` loops described previously, we consider the

following additional markovian statements:

$$\begin{array}{rcll} Stmt_M & ::= & x = \texttt{uniform}_l(e_1, e_2) & \wr\ x \in \mathcal{V}, e_1, e_2 \in Exp\ \int \\ & | & x = \texttt{bernoulli}_l() & \\ & | & \texttt{ticks}_l(e) & \wr\ e \in Exp\ \int \end{array}$$

The function $\texttt{uniform}_l(e_1, e_2)$ draws a random integer value from a discrete uniform distribution over the interval $[e_1, e_2]$, while the function $\texttt{bernoulli}_l()$ returns a boolean value according to a Bernoulli distribution. Finally, the function $\texttt{ticks}_l(e)$ models the fact that the program will spend e ticks in the current control location, which triggers a transition in the Markov chain of the program.

Using these primitive functions, we can define any markovian behavior. Since we are interested in communication protocols, we provide a number of auxiliary functions based on these primitives. For example, the function `unicast`() presented previously is equivalent to choose a random success outcome from a Bernoulli distribution to model the lossy nature of the wireless link, and perform a wait period for emulating the duration of the communication. Another example is the function `sleep`(e), which is equivalent to calling `ticks`(e) after setting the radio state to off.

5.2.1 Markovian Traces

We develop a particular stochastic semantics that is isomorphic to a discrete time Markov chain. At the bottom level of this semantics, we have the notion of random events Ξ representing the outcomes of the probability distributions generated during program execution. We can distinguish between two types of random events. The events $\mathbf{b}_l$ and $\overline{\mathbf{b}}_l$ denote the two outcomes of a statement $\texttt{bernoulli}_l()$. Also, the outcomes of the statement $\texttt{uniform}_l(e_1, e_2)$ are given by the set $\{\mathbf{u}_l^{i,a,b} \mid i \in [a, b]\}$, where a and b are the evaluation in the current execution environment of e_1 and e_2 respectively.

Naively, we can consider a Markov chain as a classical automaton over the alphabet Ξ recognizing the probabilistic traces of the program consisting in sequences of random events. However, Markov chains are not just a set of probabilistic traces, but embed a notion of time that is fundamental. Indeed, transitions in a Markov chain occur solely when at least one time tick has elapsed, since a state of the Markov chain can not have a null sojourn time. As we consider that only the `ticks`(e) statement advances time, some of the program transitions are non-observable at the time scale of the chain. This leads us to a notion of a two-level trace semantics making the distinction between observable and non-observable transitions, which has been introduced many years ago by Radhia Cousot in her thesis [Cou85, Section 2.5.4]. In the following, we give a

formal definition of these two types of traces:

Definition 5.1 (Scenarios)**.** *A sequence of non-observable transitions is called a scenario and is defined as* $\omega \in \Omega \triangleq \Xi^*$ *to denote sequences of random events that occur between two observable states. In the sequel, we denote by* ε *the empty scenario word.*

Definition 5.2 (Markovian traces)**.** *The observable markovian traces is the set* $\mathcal{T}_\Sigma^\Omega \triangleq \{\sigma_0 \overset{\omega_1}{\to} \sigma_1 \overset{\omega_2}{\to} \cdots \mid \sigma_i \in \Sigma \wedge \omega_i \in \Omega\}$ *of transitions among observable states and labeled with scenarios. An observable state is a tuple* $(l, \rho, \nu) \in \Sigma \triangleq \mathcal{L} \times \mathcal{E} \times \mathbb{N}$ *where (i)* $l \in \mathcal{L}$ *is a program location, (ii)* $\rho \in \mathcal{E} \triangleq \mathcal{V} \to \mathbb{Z}$ *is a program environment and (iii)* $\nu \in \mathbb{N}$ *is a sojourn time representing the number of ticks spent in that state.*

This notion of markovian traces is actually a set-based representation of Markov chains that fits more adequately within the framework of abstract interpretation. It allows a fluent extension of the classical trace semantics for supporting the particular stochastic and temporal features of discrete time Markov chains. In the following paragraph, we define this semantics domain and we present the most important transfer functions.

5.2.2 Semantics Domain

The concrete semantics domain of our analysis is given by:

$$\mathcal{D} \triangleq \wp(\mathcal{T}_\Sigma^\Omega \times \mathcal{E} \times \Omega)$$

An element $(\tau, \rho, \omega) \in X \subseteq \mathcal{D}$ encodes the set of traces reaching a given program location and is composed of three parts: (i) the observable trace $\tau \in \mathcal{T}_\Sigma^\Omega$ containing the past markovian transitions before the current time tick, (ii) the current memory environment $\rho \in \mathcal{E}$, and (iii) the partial scenario $\omega \in \Omega$ of non-observable random events that occurred between the last tick and the current execution moment.

To obtain the set of all traces of a program P, we proceed by induction on its abstract syntax tree using a set of concrete transfer functions $\mathbf{S}[\![.]\!] \to \mathcal{D} \to \mathcal{D}$. In Figure 5.3 we give a summary of these functions. We recall that $\mathbf{E}[\![e]\!] \in \wp(\mathcal{E}) \to \wp(\mathcal{E})$ provides the possible evaluations of an expression in a set of environments, as defined previously in (3.6). Non-probabilistic statements have a standard definition. The assignment statement updates the current memory environment by mapping the left-hand variable to the actual evaluation the expression. For the `if` assignment, we filter the current environments depending on the evaluation of the condition, and we analyze each branch independently before merging the results. Also, a loop statement is formalized as a fixpoint on the sequences of body evaluation with a filter to extract the iterations violating the loop condition.

$$\mathbf{S}[\![x = e]\!]R = \{(\tau, \rho[x \mapsto v], \omega) \mid (\tau, \rho, \omega) \in R \wedge v \in \mathbf{E}[\![e]\!]\{\rho\}\}$$
$$\mathbf{S}[\![?(e \bowtie 0)]\!]R = \{(\tau, \rho, \omega) \in R \mid \exists v \in \mathbf{E}[\![e]\!]\{\rho\} : v \bowtie 0\}$$
$$\mathbf{S}[\![\texttt{if}(e \bowtie 0)\{s_1\}\{s_2\}]\!]R = (\mathbf{S}[\![s_1]\!] \circ \mathbf{S}[\![?(e \bowtie 0)]\!]R) \cup (\mathbf{S}[\![s_2]\!] \circ \mathbf{S}[\![?(e \not\bowtie 0)]\!]R)$$
$$\mathbf{S}[\![\texttt{while}(e \bowtie 0)\{s\}]\!]R = \mathbf{S}[\![?(e \not\bowtie 0)]\!](\mathsf{lfp}\, \lambda X.\ R \cup \mathbf{S}[\![s]\!] \circ \mathbf{S}[\![?(e \bowtie 0)]\!]X)$$

$$\mathbf{S}[\![x = \texttt{bernoulli}_l()]\!]R = \{(\tau, \rho[x \mapsto b], \omega.\xi) \mid (\tau, \rho, \omega) \in R \wedge (b, \xi) \in \{(1, \mathbf{b}_l), (0, \overline{\mathbf{b}}_l)\}\}$$
$$\mathbf{S}[\![x = \texttt{uniform}_l(e_1, e_2)]\!]R =$$
$$\{(\tau, \rho[x \mapsto i], \omega.\mathbf{u}_l^{i,a,b}) \mid (\tau, \rho, \omega) \in R \wedge a \in \mathbf{E}[\![e_1]\!]\{\rho\} \wedge b \in \mathbf{E}[\![e_2]\!]\{\rho\} \wedge i \in [a, b]\}$$
$$\mathbf{S}[\![\texttt{ticks}_l(e)]\!]R = \{(\tau \xrightarrow{\omega} (l, \rho, \nu), \rho, \varepsilon) \mid (\tau, \rho, \omega) \in R \wedge \nu \in \mathbf{E}[\![e]\!]\{\rho\}\}$$

Figure 5.3: Concrete transfer functions.

The semantics of the statement $x = \texttt{bernoulli}_l()$ is to fork the current partial scenarios ω depending on the result of the function. We append the event $\mathbf{b}_l$ in the true case, or the event $\overline{\mathbf{b}}_l$ in the false case and we update the variable x with the returned value in the current memory environment. For the statement $x = \texttt{uniform}_l(e_1, e_2)$, we also fork the partial scenarios and update the variable x accordingly, but the difference is that the number of branches depend on the evaluations of e_1 and e_2 in the current memory environments. More precisely, the number of forks corresponds to the number of integer points between the values of e_1 and e_2. Note that, for these two statements, the markovian traces part is not modified since they are tick-less. This is not the case for the $\texttt{ticks}_l(e)$ statement that append the morkovian traces with a new transition to state where the sojourn time is equal to the evaluation of the expression e. The label of this new transition is simply the computed partial scenario, which is reset to the empty word ε since we keep track of events traces only between two ticks statements.

5.2.3 Stationary Distribution

After collecting the set $T \subseteq \mathcal{T}_\Sigma^\Omega$ of all possible markovian traces of the program P, we want to compute the *stationary distribution* of the associated Markov chain, which reflects the proportion of time spent in every observable state and represents the key element to obtain the desired proportional expectations. To do that, we have first to construct the square transition matrix $\mathbf{P}$ defined as[1]:

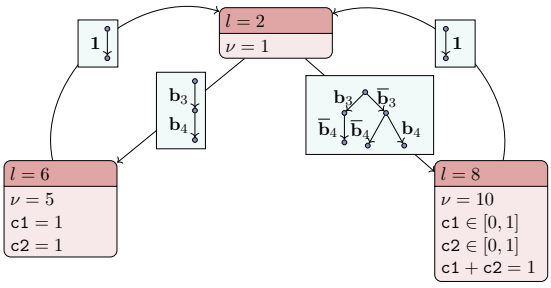
$$\mathbf{P}_{(l,\rho,\nu),(l',\rho',\nu')} \triangleq \frac{\nu'}{\nu} \sum_{(l,\rho,\nu) \xrightarrow{\omega} (l',\rho',\nu') \in T} \Pr(\omega) \tag{5.2}$$

where (l, ρ, ν) and (l', ρ', ν') are two reachable states in the traces T. The function $\Pr \in \Omega \to [0, 1]$ gives the probability of the scenarios and is computed recursively as

[1] It is worth noting that this matrix $\mathbf{P}$ is not the classical stochastic matrix of the Markov chain.

```
1  while(1) {
2    ticks_2(1);
3    c1 = bernoulli_3();
4    c2 = bernoulli_4();
5    if (c1 && c2) {
6      ticks_6(5);
7    } else {
8      ticks_8(10);
9    }
10 }
```

(a)



(b)

Figure 5.4: (a) An example probabilistic program. (b) An abstraction of observable traces represented as hierarchical automaton.

follows:

$$\Pr(\omega) \triangleq \begin{cases} 1 & \text{if } \omega = \varepsilon \\ p_l & \text{if } \omega = \mathbf{b}_l \\ 1 - p_l & \text{if } \omega = \overline{\mathbf{b}}_l \\ \frac{1}{b-a+1} & \text{if } \omega = \mathbf{u}_l^{i,a,b} \\ \prod_{1 \leq i \leq n} \Pr(\xi_i) & \text{if } \omega = \xi_1 ... \xi_n \end{cases}$$

Finally, the stationary distribution represents the eigenvector $\boldsymbol{\pi}$ of the matrix $\mathbf{P}$ associated to the eigenvalue 1, which is obtained by solving the system:

$$\boldsymbol{\pi} = \boldsymbol{\pi}\mathbf{P} \tag{5.3}$$

with the additional normalization constraint $\sum_{(l,\rho,\nu)} \boldsymbol{\pi}_{(l,\rho,\nu)} = 1$. Since the size of $\mathbf{P}$ depends on the size of the reachable states space, $\boldsymbol{\pi}$ can not be computed automatically in general. In the following, we propose a computable abstraction of Markov chains to over-approximate the traces T. Afterwards, we show how we can infer guaranteed bounds of $\boldsymbol{\pi}$ using information provided by our abstract chain.

5.3 Abstract Semantics

In order to analyze a program statically, we need a computable abstraction of the concrete semantics domain $\mathcal{D}$. The basic idea is to first partition the set of observable program states $\mathcal{L} \times \mathcal{E} \times \mathbb{N}$ with respect to the program locations, resulting in the intermediate abstraction $\mathcal{L} \times \wp(\mathcal{E} \times \mathbb{N})$. For each location, the set of associated environments is then abstracted with a stock numerical domain $\mathcal{E}^\sharp$, by considering the sojourn time as a program variable ν. Consequently, we obtain the abstract states domain $\Sigma^\sharp \triangleq \mathcal{L} \times \mathcal{E}^\sharp$.

As a consequence of this partitioning, observable states at the same program lo-

cation will be merged into a single abstract state. It is important to notice that, at the trace level, we may have many scenarios connecting two successive abstract states. We illustrate this fact in Figure 5.4 depicting a simple probabilistic program and an automata representation of its traces by applying the previous partitioning. Between locations 2 and 6, only the events word $\mathbf{b}_3\mathbf{b}_4$ can occur since this corresponds to the true branch of the test statement at line 5. On the other hand, the transition between locations 2 and 8 is labeled with events words $\{\mathbf{b}_3\overline{\mathbf{b}}_4, \overline{\mathbf{b}}_3\overline{\mathbf{b}}_4, \overline{\mathbf{b}}_3\mathbf{b}_4\}$.

The presence of these multi-words transitions leads to a *hierarchical automata* structure organized in two levels. On one hand, an automata structure is used to encode the transitions between observable abstract states. On the other hand, and for each observable transition, another automata structure is used to encode the set of events words (i.e. scenarios) connecting the endpoints of the transition. In other words, we abstract markovian traces with an automation which transitions have also an automata representing a set of scenarios. For modularity reasons, we present however a single generic automata domain to represent regular languages over any abstract alphabet. Afterwards, we instantiate two automata-based domains for abstracting events words and markovian traces. For the sake of clarity, we only define the operators and we postpone the examples illustrating them when instantiating the domains.

5.3.1 Abstract Automata

Legall et al. proposed a lattice automata domain [LGJ07] to represent words over an abstract alphabet having a lattice structure. We extend this domain to support also abstraction at the state level, which is important to approximate markovian traces. In the following, we describe the structure of our domain and we propose some important operators:

Domain functor. An instance abstract automata domain is given by the functor construction $\mathbf{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp)$. The parameter domain $\mathfrak{A}^\sharp$ is an abstraction of some concrete alphabet symbols $\mathfrak{A}$ and is assumed to have a concretization function $\gamma_{\mathfrak{A}} \in \mathfrak{A}^\sharp \to \wp(\mathfrak{A})$, a partial order $\sqsubseteq_{\mathfrak{A}}$, a join operator $\sqcup_{\mathfrak{A}}$, a meet operator $\sqcap_{\mathfrak{A}}$ and a least element $\bot_{\mathfrak{A}}$. The second parameter domain $\mathfrak{S}^\sharp$ is an abstraction of some concrete states $\mathfrak{S}$ and is equipped with a concretization function $\gamma_{\mathfrak{S}} \in \mathfrak{S}^\sharp \to \wp(\mathfrak{S})$, a partial order $\sqsubseteq_{\mathfrak{S}}$, a join operator $\sqcup_{\mathfrak{S}}$, an equivalence relation $\equiv_{\mathfrak{S}}$, a least element $\bot_{\mathfrak{S}}$ and a widening operator $\triangledown_{\mathfrak{S}}$.

An instance $\mathbf{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp)$ provides a representation of traces composed of successive transitions among abstract states and labeled with symbols from the abstract alphabet. It has a standard automata structure defined as the tuple $(S, s_0^\sharp, F, \Delta)$, where:

- $S \subseteq \mathfrak{S}^\sharp$ is the set of states and $s_0^\sharp \in S$ is the initial state.
- $F \subseteq S$ is the set of final states.
- $\Delta \subseteq S \times \mathfrak{A}^\sharp \times S$ is the transition relation.

In the following, we will denote by $A = (S, s_0^\sharp, F, \Delta)$, $A_1 = (S_1, s_{0_1}^\sharp, F_1, \Delta_1)$ and $A_2 = (S_2, s_{0_2}^\sharp, F_2, \Delta_2)$ as three abstract automata instances of $\mathbf{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp)$. We also define the auxiliary functions $\mathbf{L} \in \mathbf{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp) \to \mathfrak{A}^{\sharp\star}$ and $\mathbf{T} \in \mathbf{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp) \to \wp(\mathcal{T}_{\mathfrak{S}^\sharp}^{\mathfrak{A}^\sharp})$ giving the set of accepted words and traces respectively.

Append. We introduce the append operator $\odot_\phi \in \mathbf{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp) \times \mathfrak{A}^\sharp \to \mathbf{A}(\mathfrak{A}^\sharp, \mathfrak{S}^\sharp)$ that extend an abstract automaton with a set of new leave transitions labeled with the given abstract alphabet symbol. From every final state $s_i^\sharp$, a new edge is created to a new state computed as the image of $s_i^\sharp$ through the transfer function $\phi \in \mathfrak{S}^\sharp \to \mathfrak{S}^\sharp$ that annotates the operator $odot_\phi$. This operator can be formulated as follows:

$$
\begin{array}{l}
A \odot_\phi a^\sharp \triangleq \\
\quad \mathsf{let}\ F' = \{\phi(s^\sharp) \mid s^\sharp \in F\}\ \mathsf{in} \\
\quad (S \cup F', s_0^\sharp, F', \Delta \cup \{s^\sharp \xrightarrow{a^\sharp} \phi(s^\sharp) \mid s^\sharp \in F\})
\end{array}
$$

Product. The product of two automata A_1 and A_2, denoted by $A_1 \otimes A_2$, is the set of tuple transitions $(s_1, s_2) \xrightarrow{a^\sharp} (s_1', s_2') \in \mathfrak{S}^{\sharp 2} \times \mathfrak{A}^\sharp \times \mathfrak{S}^{\sharp 2}$ that encodes the simultaneous traversal on A_1 and A_2. Basically, the product allows us to track the state of both automata when going along the same trace, which is helpful for checking inclusion between A_1 and A_2 or for merging them, as we will see shortly.

We give in Figure 5.5 the algorithm of this operator. Recursively, and starting from the initial states $(s_{0_1}^\sharp, s_{0_2}^\sharp)$, we first compute the individual next transitions of $s_1^\sharp$ and $s_2^\sharp$ respectively (lines 7 – 8). After that, we need to compare these individual transitions in order know whether both automata behaves similarly when reaching $(s_1^\sharp, s_2^\sharp)$ or not. However, since these transitions are labeled with lattice elements, the comparison between two transition symbols can not be resolved in every situation. More particularly, when lacking of a complement operator for the alphabet domain $\mathfrak{A}^\sharp$, we can not derive precisely the transitions that only one automaton can go through.

A solution is to relax the comparison condition by first considering that we can move to states $(q_1^\sharp, q_2^\sharp)$ if the intersection between the individual transition symbols is not empty. If this is the case, we consider that this product transition is labeled with the union of these symbols (line 9). This over-approximation may be coarse in some

Input : Two abstract automata A_1 and A_2
Output : $A_1 \otimes A_2 \in \wp(\mathfrak{S}^{\sharp^2} \times \mathfrak{A}^\sharp \times \mathfrak{S}^{\sharp^2})$

1 $wq \leftarrow \{(s^\sharp_{0_1}, s^\sharp_{0_2})\}$;
2 $A_1 \otimes A_2 \leftarrow \varnothing$;
3 $H \leftarrow \varnothing$;
4 **while** $wq \neq \varnothing$ **do**
5 $\quad (s^\sharp_1, s^\sharp_2) \leftarrow \mathtt{pop}(wq)$;
6 $\quad H \leftarrow H \cup \{(s^\sharp_1, s^\sharp_2)\}$;
$\quad$ *Compute individual next transitions*
7 $\quad \delta_1 \leftarrow \{(a^\sharp, q^\sharp_1) \mid s^\sharp_1 \xrightarrow{a^\sharp} q^\sharp_1 \in \Delta_1\}$;
8 $\quad \delta_2 \leftarrow \{(a^\sharp, q^\sharp_2) \mid s^\sharp_2 \xrightarrow{a^\sharp} q^\sharp_2 \in \Delta_2\}$;
$\quad$ *Compute common transitions*
9 $\quad \delta_\sqcap \leftarrow \{(a^\sharp, (q^\sharp_1, q^\sharp_2)) \mid (a^\sharp_1, q^\sharp_1) \in \delta_1 \wedge (a^\sharp_2, q^\sharp_2) \in \delta_2 \wedge a^\sharp_1 \sqcap_{\mathfrak{A}} a^\sharp_2 \neq \bot_{\mathfrak{A}} \wedge a^\sharp = a^\sharp_1 \sqcup_{\mathfrak{A}} a^\sharp_2\}$;
$\quad$ *Compute singular transitions*
10 $\quad \delta_{\bot_1} \leftarrow \{(a^\sharp, q^\sharp_1) \in \delta_1 \mid \forall(-, (q^\sharp, -)) \in \delta_\sqcap : q^\sharp_1 \neq q^\sharp\}$;
11 $\quad \delta_{\bot_2} \leftarrow \{(a^\sharp, q^\sharp_2) \in \delta_2 \mid \forall(-, (q^\sharp, -)) \in \delta_\sqcap : q^\sharp_2 \neq q^\sharp\}$;
$\quad$ *Construct new product transitions from $\delta_\sqcap$, $\delta_{\bot_1}$ and $\delta_{\bot_2}$*
12 $\quad \Delta \leftarrow \{(s^\sharp_1, s^\sharp_2) \xrightarrow{a^\sharp} (q^\sharp_1, q^\sharp_2) \mid (a^\sharp, (q^\sharp_1, q^\sharp_2)) \in \delta_\sqcap\} \cup$
$\qquad \{(s^\sharp_1, s^\sharp_2) \xrightarrow{a^\sharp} (q^\sharp_1, \bot_{\mathfrak{A}}) \mid (a^\sharp, q^\sharp_1) \in \delta_{\bot_1}\} \cup$
$\qquad \{(s^\sharp_1, s^\sharp_2) \xrightarrow{a^\sharp} (\bot_{\mathfrak{A}}, q^\sharp_1) \mid (a^\sharp, q^\sharp_2) \in \delta_{\bot_2}\}$;
$\quad A_1 \otimes A_2 \leftarrow \Delta \cup A_1 \otimes A_2$;
13 $\quad wq \leftarrow wq \cup \{x \mid \exists(-, -, x) \in \Delta\} \setminus H$;
14 **end**

Figure 5.5: Product operator algorithm.

situations, but it ensures the soundness of the computations. Afterward, when there exists an individual transition symbol, let say to state $q^\sharp_1$ in A_1, that does not intersect with any other transition of the other automaton, we can conclude that only A_1 can perform a transition from $(s^\sharp_1, s^\sharp_2)$ with this symbol. We call such transition a singular transition (lines 10 – 11) and we encode it as a product move to $(q^\sharp_1, \bot_{\mathfrak{S}})$ (line 12).

Order. Using the product operator, we define an order relation $\sqsubseteq_{\mathbf{A}}$ that allows us to check if the accepted traces of one automata A_1 is included in the traces of another one A_2. To verify this, we simply examine the reachable tuples in the product transition relation $A_1 \otimes A_2$, and we check that every final state of A_1 is always coupled with a final state of A_2. This means that all accepted traces by A_1 are also accepted by A_2:

$$\begin{aligned} A_1 \sqsubseteq_{\mathbf{A}} A_2 \Leftrightarrow \quad & \forall (s^\sharp_1, s^\sharp_2) \xrightarrow{a^\sharp} (q^\sharp_1, q^\sharp_2) \in A_1 \otimes A_2 : \\ & s^\sharp_1 \sqsubseteq_{\mathfrak{S}} s^\sharp_2 \wedge q^\sharp_1 \sqsubseteq_{\mathfrak{S}} q^\sharp_2 \wedge \\ & (s^\sharp_1 \in F_1 \Rightarrow s^\sharp_2 \in F_2) \wedge (q^\sharp_1 \in F_1 \Rightarrow q^\sharp_2 \in F_2) \end{aligned}$$

Join. Joining two abstract automata A_1 and A_2 results in a new automaton A accepting, at least, the traces of A_1 and the traces of A_2. The construction of the join

Input : Two abstract automata A_1 and A_2
Output : A widened abstract automata $A = A_1 \triangledown_{\mathbf{A}} A_2$

1 $A = (S, s_0, F, \Delta) \leftarrow A_1$;
⟅ Find the increment transitions ⟆
2 $\delta \leftarrow \{(s_1^\sharp, s_2^\sharp \xrightarrow{a^\sharp} q_2^\sharp) \mid (s_1^\sharp, s_2^\sharp) \xrightarrow{a^\sharp} (\bot_{\mathfrak{S}}, q_2^\sharp) \in A_1 \otimes_{\mathbf{A}} A_2\}$;
3 **while** $\delta \neq \varnothing$ **do**
4 $(s_1^\sharp, s_2^\sharp, q_2^\sharp, a^\sharp) \leftarrow \texttt{choose}(\delta)$;
⟅ Search in A for equivalent states to $q_2^\sharp$ ⟆
5 $Q_\equiv \leftarrow \{s^\sharp \in S \mid s^\sharp \equiv_Q q_2^\sharp\}$;
6 **if** $Q_\equiv = \varnothing$ **then**
⟅ Add $q_2^\sharp$ as a new equivalent state ⟆
7 $q_\equiv^\sharp \leftarrow q_2^\sharp$;
8 **else**
9 $q_\equiv^\sharp \leftarrow \texttt{sort}(Q_\equiv)$;
10 **end**
⟅ Construct a new automaton with the missing transition. The notation $(c)?a : b$ denotes the expression that returns a is case c is verified or b otherwise ⟆
11 $A \leftarrow (S \cup q_\equiv^\sharp, s_0, F \cup (q_\equiv^\sharp \in F_2?\{q_\equiv^\sharp\} : \varnothing), \Delta \cup \{(s_1^\sharp, a^\sharp, q_\equiv^\sharp)\})$;
12 $\delta \leftarrow \{(s_1^\sharp, s_2^\sharp \xrightarrow{a^\sharp} q_2^\sharp) \mid ((s_1^\sharp, s_2^\sharp) \xrightarrow{a^\sharp} (\bot_{\mathfrak{S}}, q_2^\sharp)) \in A \otimes_{\mathbf{A}} A_2\}$;
13 **end**

Figure 5.6: Widening operator algorithm.

automaton A is also provided through the computation of the product transitions as follows:

$$\begin{cases} S = \{s_1^\sharp \sqcup_{\mathfrak{S}} s_2^\sharp \mid (s_1^\sharp, s_2^\sharp) \xrightarrow{-} - \in A_1 \otimes A_2\} \\ s_0^\sharp = s_{0_1}^\sharp \sqcup_{\mathfrak{S}} s_{0_2}^\sharp \\ F = \{s_1^\sharp \sqcup_{\mathfrak{S}} s_2^\sharp \mid (s_1^\sharp, s_2^\sharp) \xrightarrow{-} - \in A_1 \otimes A_2 \wedge s_1^\sharp \in F_1 \vee s_2^\sharp \in F_2\}) \\ \Delta = \{(s_1^\sharp \sqcup_{\mathfrak{S}} s_2^\sharp) \xrightarrow{a^\sharp} (q_1^\sharp \sqcup_{\mathfrak{S}} q_2^\sharp) \mid (s_1^\sharp, s_2^\sharp) \xrightarrow{a^\sharp} (q_1^\sharp, q_2^\sharp) \in A_1 \otimes A_2\} \end{cases}$$

which means that we simply map each reachable product state to the join of its individual states and that final states are the image of the products states containing at least one final state.

Widening. When analyzing a program loop, an abstract automaton can grow indefinitely, so we need to extrapolate its growth pattern in a way to ensure termination in finite time. For this reason, several automata widening algorithms have been proposed that can be categorized into two families. The first family [Vil02] compares the result of two successive iterations of the loop and tries to detect the increment transitions in order to wrap it and create cycles extrapolating the iterations. The second family [Fer01a, LGJ07] joins the two automata and applies a bisimulation-based minimization that merges similar states by comparing their transitions at some given depth.

In our case, we use a hybrid approach combining aspects from both techniques,

as detailed in Figure 5.6. Assume that A_1 and A_2 are the results of two successive iterations. Without loss of generality, we assume that $A_1 \sqsubseteq_{\mathbf{A}} A_2$. As in [Vil02], we compare A_1 and A_2 in order to extract the increment transition (line 2). After that, we will modify A_1 in a way to eliminate all increment transitions by extrapolating them. To do that, we choose one increment transition (line 4), denoted by $(s_1^\sharp, s_2^\sharp \xrightarrow{a^\sharp} q_2^\sharp)$ which means that A_1 at state $s_1^\sharp$ can not recognize the symbol $a^\sharp$ while A_2 recognizes it through a move from $s_2^\sharp$ to $q_2^\sharp$. We need now to choose an equivalent state to $q_2^\sharp$ in A_1 in order to add the missing transition. In the widening operator presented in [Vil02], any state of A_1 in the predecessors of $s_1^\sharp$ can be chosen. However, this non-determinism leads to imprecise results. We propose a more meaningful selection of equivalent states based on a similarity distance. The basic idea is to sort states in A_1 depending on how they compare to the missing state $q_2^\sharp$. The comparison is performed with a Jaccard based distance for expressing the proportion of common partial traces that a state shares with $q_2^\sharp$. The comparison is limited however to a predefined depth constant. After selecting the most comparable equivalent state $q_\equiv^\sharp$, we add the missing transition $s_1^\sharp \xrightarrow{a^\sharp} q_\equiv^\sharp$ to A_1 (line 11), and we iterate the same process until resolving all the differences.

5.3.2 Abstract Scenarios

Using the functor domain **A**, we can now instantiate an abstract scenario domain for approximating words of random events. Two considerations are important to take into account. First, the length of these words may depend on some variables of the program. It is clear that ignoring these relations may lead to imprecise computations of the stationary distribution. Consequently, we enrich the domain with an abstract Parikh vector to count the number of occurrences of random event within accepted words. To preserve some relationships with program variables, we use a relational numerical domain, such as octagons or polyhedra.

The second consideration is related to the uniform distribution. As shown previously in the concrete transfer function in Figure 5.3, the number of outcomes depends on the bounds provided as argument to the function `uniform`. Since these arguments are evaluated in the running environment, we can have an infinite number of outcomes at a given control location when considering all possible executions. Therefore, we perform a simplifying abstraction on the random events Ξ in order to obtain a finite size alphabet and avoid such explosion. We partition the outcomes of each uniform distribution into a fixed number U of abstract outcomes, where U is a parameter of the analysis. As a consequence, we build the finite set of abstract events $\Xi^\sharp \triangleq \{\mathbf{b}_l, \overline{\mathbf{b}}_l \in \Xi\} \cup \{\boldsymbol{\mu}_l^i \mid \mathbf{u}_l^{j,a,b} \in \Xi \wedge 1 \leq i \leq U\}$. For the Parikh vector, we associate

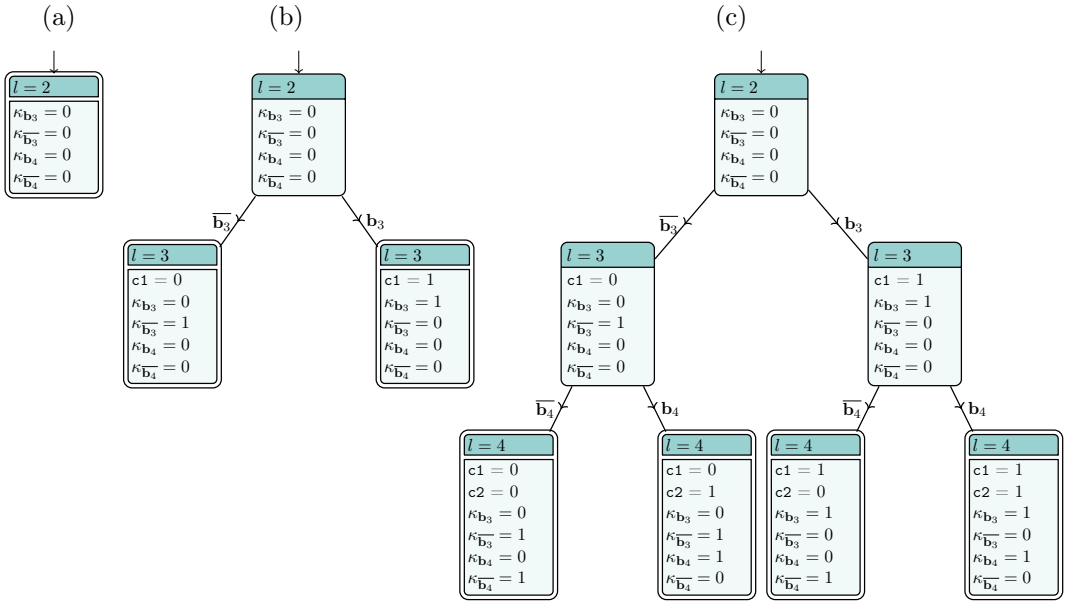


Figure 5.7: Examples of executing the abstract transfer function for bernoulli statement. (a) Initial empty scenario. (b) Execution of $\texttt{c1} = \texttt{bernoulli}_3()$. (b) Execution of $\texttt{c2} = \texttt{bernoulli}_4()$.

to every abstract event $\xi^\sharp \in \Xi^\sharp$ a counter variable $\kappa_{\xi^\sharp} \in \mathbb{N}$ that will be incremented whenever the event $\xi^\sharp$ occurs.

Therefore, we define the domain of abstract scenarios as $\Omega^\sharp = \mathbf{A}(\wp(\Xi^\sharp), \Sigma^\sharp)$ where $\Sigma^\sharp$ is the mapping $\mathcal{L} \to \mathcal{E}^\sharp$ from program locations to a stock numeric abstract domain. Let us now describe how statements affect an abstract scenario. Consider the statement $\texttt{ticks}_l(e)$. Since scenarios are accumulated between two ticks, this statement will reset the partial scenarios to a new empty automaton $\varepsilon^\sharp$ containing a null Parkih vector:

$$\mathbf{S}[\![\texttt{ticks}_l(e)]\!]^\sharp_\Omega \omega^\sharp \triangleq \varepsilon^\sharp$$

For the $\texttt{bernoulli}_l()$ statement, we create two new transitions labeled with the abstract events $\mathbf{b}_l$ and $\overline{\mathbf{b}}_l$ respectively and we update the Parikh vector accordingly, as shown by the following:

$$\begin{aligned}
&\mathbf{S}[\![x = \texttt{bernoulli}_l()]\!]^\sharp_\Omega \omega^\sharp \triangleq \\
&\quad \text{let } \phi_0(l', \rho^\sharp) = (l, \mathbf{S}[\![x = 0]\!]^\sharp_\mathcal{E} \circ \mathbf{S}[\![\kappa_{\overline{\mathbf{b}}_l} + +]\!]^\sharp_\mathcal{E} \rho^\sharp) \text{ in} \\
&\quad \text{let } \phi_1(l', \rho^\sharp) = (l, \mathbf{S}[\![x = 1]\!]^\sharp_\mathcal{E} \circ \mathbf{S}[\![\kappa_{\mathbf{b}_l} + +]\!]^\sharp_\mathcal{E} \rho^\sharp) \text{ in} \\
&\quad \omega^\sharp \odot_{\phi_0} \{\overline{\mathbf{b}}_l\} \sqcup_\mathbf{A} \omega^\sharp \odot_{\phi_1} \{\mathbf{b}_l\}
\end{aligned}$$

Example 5.3. Consider the previous probabilistic program shown in Figure 5.4(a).

After executing the `ticks` statement at location 2, we obtain an empty abstract scenario shown in Figure 5.7(a). Next, the first `bernoulli` statement at location 3 creates a fork to two new states as shown in Figure 5.7(b). Notice that the transfer function updates both the program variable and Parikh counters. The next statement will create two additional forks, one for each final state, as shown in Figure 5.7(c). When executing the `if` at line 5, the different branches can be easily determined by applying the filter on the final states environments. For example, when choosing the true branch, only one final state verifies the condition `c1 && c2`, which corresponds to the scenario word $\mathbf{b}_3\mathbf{b}_4$. ○

Similarly, we give the following abstract transfer function for the $\texttt{uniform}_l(e_1, e_2)$ statement that generates U new transitions with appropriate state updates:

$$\begin{aligned}&\mathbf{S}[\![x = \texttt{uniform}_l(e_1, e_2)]\!]^\sharp_\Omega \omega^\sharp \triangleq \\ &\quad \text{let } \phi(i) = \lambda\ (l', \rho^\sharp).\ (l, \mathbf{S}[\![x = [e_1 + \tfrac{(i-1)(e_2-e_1+1)}{U}, e_1 + \tfrac{i(e_2-e_1+1)}{U}[\!]^\sharp_{\mathcal{E}} \circ \mathbf{S}[\![\kappa_{\boldsymbol{\mu}_l^i} + +]\!]^\sharp_{\mathcal{E}} \rho^\sharp) \text{ in} \\ &\quad \bigsqcup_{1 \le i \le U} \omega^\sharp \odot_{\phi(i)} \{\boldsymbol{\mu}_l^i\}\end{aligned}$$

5.3.3 Abstract Markov Chains

The product $\mathcal{D}^\sharp \triangleq \mathcal{T}^\sharp \times \Omega^\sharp$ defines our Abstract Markov Chains domain, which is also the abstract semantics domain.It is composed of two parts. The first one is an abstraction of the markovian traces and is defined as the instance automata domain $\mathcal{T}^\sharp \triangleq \mathbf{A}(\Omega^\sharp, \Sigma^\sharp)$. This automaton is used to approximate the set of past observable traces reaching a given program location. The second part is an abstraction of the current partial scenarios starting from the last `ticks` statement. Since final states of an abstract scenario automaton already embeds an abstraction of the program environments, we also employ this part to encode the current environments. Let us define the abstract transfer function for $\mathcal{D}^\sharp$. When analyzing a tick-less statement s, only the partial scenario part is affected:

$$\mathbf{S}[\![s]\!]^\sharp(T^\sharp, \omega^\sharp) \triangleq (T^\sharp, \mathbf{S}[\![s]\!]^\sharp_\Omega \omega^\sharp)$$

However, the $\texttt{ticks}_l(e)$ statement modifies both parts since the statements indicates that a new observable state has been encountered and that the pending scenarios are no longer partial and should be used to label the new transition, which can be formulated as follows:

$$\begin{aligned}&\mathbf{S}[\![x = \texttt{ticks}_l(e)]\!]^\sharp(T^\sharp, \omega^\sharp) \triangleq \\ &\quad \text{let } \phi(l', \rho^\sharp) = (l, \mathbf{S}[\![\nu = e]\!]^\sharp_{\mathcal{E}} \rho^\sharp)) \text{ in} \\ &\quad (T^\sharp \odot_\phi \omega^\sharp, \mathbf{S}[\![x = \texttt{ticks}_l(e)]\!]^\sharp_\Omega \omega^\sharp)\end{aligned}$$

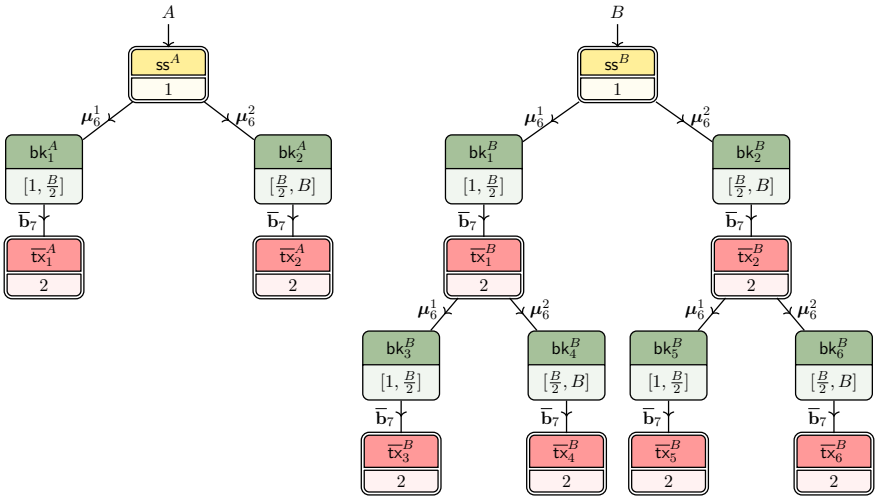
Finally, one of the most important operators for this domain is the widening operator that allows obtaining finite size abstractions of unbounded sets of markovian traces. Previously, we have given its generic algorithm and in the following, we illustrate the different steps through a detailed example.

Example 5.4. Consider our motivating example listed in Figure 5.1(a), and more particularly the inner `while` loop at line 4. Let us assume that the uniform partition parameter U is set to 2. For the sake of clarity, the observable abstract states are represented with only two fields: a unique code name and the interval of the sojourn time.

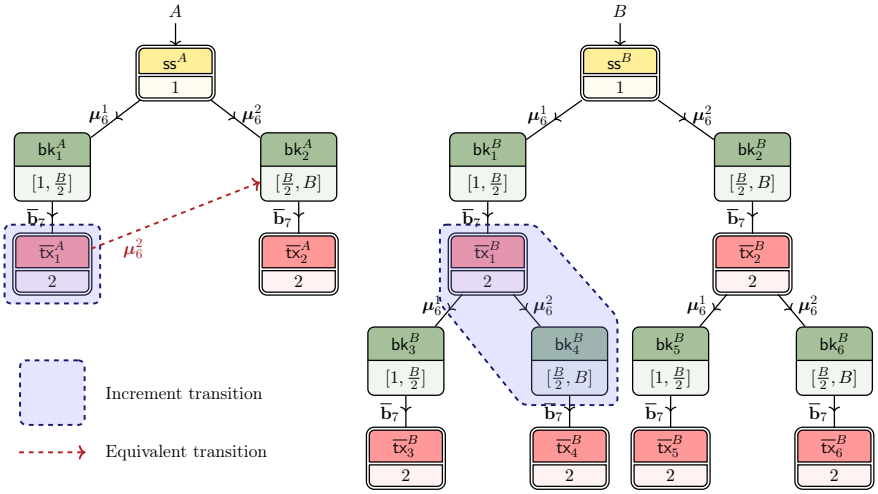
We show in Figure 5.8(a) the two automata A and B resulting from two successive loop iterations. We can notice that $A \sqsubseteq_{\mathbf{A}} B$ and that these results will keep growing indefinitely since the loop is infinite when no acknowledgment is received. To extrapolate the computations, we apply the widening operator. When comparing A and B, we can remark that a pattern appears. At each iteration, each unsuccessful transmission abstract state $\overline{\mathsf{tx}}_i^B$ is followed by two abstract backoff states with transitions labeled with the events $\boldsymbol{\mu}_6^1$ and $\boldsymbol{\mu}_6^2$ respectively. In each branch, a new unsuccessful transmission occurs, which indicates an eventual cycle.

The detection of this pattern is depicted in Figure 5.8(b) and is performed in several steps. The first one is to detect an increment transition between A and B that makes the two automata different, which corresponds to lines 2 and 12 of the widening algorithm shown in Figure 5.6. Since we have different dissimilarities, we choose an arbitrary one and the others will be processed during future iterations. In this example, we choose the increment transition $(\overline{\mathsf{tx}}_1^A, \overline{\mathsf{tx}}_1^B \xrightarrow{\boldsymbol{\mu}_6^2} \mathsf{bk}_4^B)$, which means that there exists a common path in A and B that leads to $\overline{\mathsf{tx}}_1^A$ and $\overline{\mathsf{tx}}_1^B$ respectively, but B can continue the path by recognizing the event $\boldsymbol{\mu}_6^2$ while A can not. Therefore, we need to add the missing transition $\overline{\mathsf{tx}}_1^A \xrightarrow{\boldsymbol{\mu}_6^2} ?$ to A by choosing a state equivalent enough to bk_4^B. By applying the Jaccard distance function, we choose bk_2^A as the nearest state and we add the equivalent transition to A.

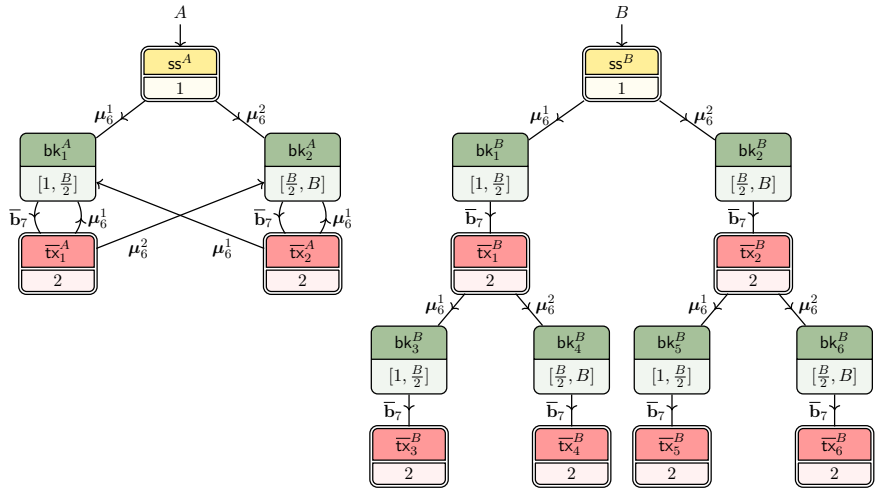
By applying the same process until no increment transition remains, we obtain the widened automaton shown in Figure 5.8(c) that encodes correct pattern. Similarly, if we continue the analysis and apply the same widening on the outer loop while eliminating redundant states, we obtain the abstract Markov chain of the program previously shown in Figure 5.2(a). ○



(a) The results of two successive loop iterations.



(b) Detection of an increment transitions.



(c) Final widened automaton.

Figure 5.8: Iterations of the widening operator.

5.4 Proportional Expectations

As described previously, the computation of proportional expectations is mostly based on finding the stationary distribution. In this section, we present a method for extracting approximate and safe bounds of the stationary distribution of a program using only information embedded in its abstract Markov chain. Let $T^\sharp = (S, s_0^\sharp, F, \Delta)$ be the markovian traces part of the program's abstract Markov chain over-approximating the set $T \subseteq \mathcal{T}_\Sigma^\Omega$ of concrete markovian traces. We begin with some preliminary definitions.

Definition 5.3. *The probability of an abstract scenario $\omega^\sharp \in \Omega^\sharp$ is defined as:*

$$\hat{\Pr}(\omega^\sharp) \triangleq \begin{cases} 1 & \text{if } \omega^\sharp = \varepsilon^\sharp \\ p_l & \text{if } \omega^\sharp = \mathbf{b}_l \\ 1 - p_l & \text{if } \omega^\sharp = \overline{\mathbf{b}}_l \\ \frac{1}{U} & \text{if } \omega^\sharp = \boldsymbol{\mu}_l^i \\ \sum_{X_1 \ldots X_n \in \mathbf{L}(\omega^\sharp)} \prod_{i=1}^n \sum_{\xi^\sharp \in X_i} \hat{\Pr}(\xi^\sharp) & \text{if } \omega^\sharp = X_1 \ldots X_n \end{cases}$$

The following important result establishes a link between the concrete and abstract scenario probabilities:

Lemma 5.1. *Given two abstract states $\sigma_i^\sharp, \sigma_j^\sharp \in S$, we have:*

$$\sum_{\sigma_i^\sharp \xrightarrow{\omega^\sharp} \sigma_j^\sharp \in \Delta} \hat{\Pr}(\omega^\sharp) \geq \max_{\sigma_i^k \in \gamma_\Sigma(\sigma_i^\sharp)} \sum_{\substack{\sigma_j^l \in \gamma_\Sigma(\sigma_j^\sharp) \\ \sigma_i^k \xrightarrow{\omega} \sigma_j^l \in T}} \Pr(\omega)$$

Also, we need a way to quantify the sojourn time elapsed within an abstract state, which is given by:

Definition 5.4. *The maximal sojourn time of an abstract state $\sigma^\sharp$ is defined as:*

$$\hat{\nu}_{\max}(\sigma^\sharp) \triangleq \max\{\nu \mid (-, -, \nu) \in \gamma_\Sigma(\sigma^\sharp)\}$$

In a similar way, we define the minimal sojourn time $\hat{\nu}_{\min}(\sigma^\sharp)$.

Since $\Sigma^\sharp$ is a mapping from program locations to a numeric abstract environment (where sojourn time ν is handled as a program variable), such boundary values can easily be computed by the numeric abstract domain. In addition, if this domain is relational, these bounds can be expressed in terms of other program variables/parameters, which is an important feature of the analysis and allows obtaining parametric solutions as we will exemplify later.

The following definition presents an abstract version of the transition matrix $\mathbf{P}$ that uses the previous quantitative forms of scenarios and sojourn times:

Definition 5.5. *The abstract transition matrix* $\hat{\mathbf{P}}$ *is a square matrix of size* $|S| \times |S|$ *where the entry for every abstract states* $\sigma_i^\sharp, \sigma_j^\sharp \in S$ *is defined as:*

$$\hat{\mathbf{P}}_{\sigma_i^\sharp, \sigma_j^\sharp} \triangleq \frac{\hat{\nu}_{\max}(\sigma_j^\sharp)}{\hat{\nu}_{\min}(\sigma_i^\sharp)} \sum_{\sigma_i^\sharp \xrightarrow{\omega^\sharp} \sigma_j^\sharp \in \Delta} \hat{\Pr}(\omega^\sharp)$$

Our goal is to infer quantitative information about the concrete distribution $\boldsymbol{\pi}$. However, since the size of this vector is possibly infinite, we propose rather to compute an abstract distribution $\hat{\boldsymbol{\pi}}$ that gives the proportion of time spent in every abstract state $\sigma^\sharp \in S$ and is defined as:

Definition 5.6. *The abstract stationary distribution* $\hat{\boldsymbol{\pi}}$ *is a vector of size* S *defined as:*

$$\hat{\boldsymbol{\pi}}_{\sigma^\sharp} \triangleq \sum_{(l,\rho,\nu) \in \gamma_\Sigma(\sigma^\sharp)} \boldsymbol{\pi}_{(l,\rho,\nu)}, \forall \sigma^\sharp \in S$$

Since spurious states in an abstract state $\gamma_\Sigma(\sigma^\sharp)$ have a null probability, the abstract stationary probability $\hat{\boldsymbol{\pi}}_{\sigma^\sharp}$ represents the exact sum of the stationary probabilities of the concrete states abstracted by $\sigma^\sharp$. Therefore, any lower and/or upper bounds that can be found about $\hat{\boldsymbol{\pi}}_{\sigma^\sharp}$ are also valid for the concrete states abstracted by $\sigma^\sharp$. To compute such bounds, we use the abstract transition matrix $\hat{\mathbf{P}}$ with the following important result:

Theorem 5.1. $\hat{\boldsymbol{\pi}} \leq \hat{\boldsymbol{\pi}}\hat{\mathbf{P}}$.

Proof. Let $\sigma_i^\sharp \in S$ an abstract state of $T^\sharp$. Using Definition 5.6 we have:

$$\begin{aligned}
\hat{\boldsymbol{\pi}}_{\sigma_i^\sharp} &= \sum_{\sigma_{i_k} \in \gamma_\Sigma(\sigma_i^\sharp)} \boldsymbol{\pi}_{\sigma_{i_k}} \\
&\wr \textit{ Using equation (5.3) } \int \\
&= \sum_{\sigma_j^\sharp \in S} \sum_{\substack{\sigma_{i_k} \in \gamma_\Sigma(\sigma_i^\sharp) \\ \sigma_{j_l} \in \gamma_\Sigma(\sigma_j^\sharp)}} \mathbf{P}_{\sigma_{j_l}, \sigma_{i_k}} \boldsymbol{\pi}_{\sigma_{j_l}} \\
&\wr \textit{ Using equation (5.2) } \int \\
&= \sum_{\sigma_j^\sharp \in S} \sum_{\substack{\sigma_{i_k} \in \gamma_\Sigma(\sigma_i^\sharp) \\ \sigma_{j_l} \in \gamma_\Sigma(\sigma_j^\sharp)}} \frac{\nu_{i_k}}{\nu_{j_l}} \boldsymbol{\pi}_{\sigma_{j_l}} \sum_{\sigma_{j_l} \xrightarrow{\omega} \sigma_{i_k} \in T} \Pr(\omega)
\end{aligned}$$

$$
\begin{aligned}
&\wr \textit{ Using Definition 5.4 } \int \\
&\leq \sum_{\sigma_j^\sharp \in S} \frac{\hat{\nu}_{\max}(\sigma_i^\sharp)}{\hat{\nu}_{\min}(\sigma_j^\sharp)} \sum_{\sigma_{j_l} \in \gamma_\Sigma(\sigma_j^\sharp)} \boldsymbol{\pi}_{\sigma_{j_l}} \sum_{\substack{\sigma_{i_k} \in \gamma_\Sigma(\sigma_i^\sharp) \\ \sigma_{j_l} \xrightarrow{\omega} \sigma_{i_k} \in T}} \Pr(\omega) \\
&\wr \textit{ Using Lemma 5.1 } \int \\
&\leq \sum_{\sigma_j^\sharp \in S} \frac{\hat{\nu}_{\max}(\sigma_i^\sharp)}{\hat{\nu}_{\min}(\sigma_j^\sharp)} \sum_{\sigma_j^\sharp \xrightarrow{\omega^\sharp} \sigma_i^\sharp \in \Delta} \hat{\Pr}(\omega^\sharp) \sum_{\sigma_{j_l} \in \gamma_\Sigma(\sigma_j^\sharp)} \boldsymbol{\pi}_{\sigma_{j_l}} \\
&\wr \textit{ Using Definition 5.5 } \int \\
&= \sum_{\sigma_j^\sharp \in S} \hat{\mathbf{P}}_{\sigma_j^\sharp, \sigma_i^\sharp} \hat{\boldsymbol{\pi}}_{\sigma_j^\sharp}
\end{aligned}
$$

□

Example 5.5. Let us consider again our motivating example to explain how this theorem allows us to obtain the linear parametric system (5.1). The first step is to construct the abstract transition matrix $\hat{\mathbf{P}}$. To do so, we parse the structure of the abstract Markov chain shown in Figure 5.2(a) to retrieve the bounds of sojourn time for every abstract state and the probability measure for every abstract scenario. By applying Definition 5.5 and considering that the rows and columns of the matrix $\hat{\mathbf{P}}$ are ordered w.r.t to $(\mathsf{ss}, \mathsf{bk}_1, \mathsf{bk}_2, \overline{\mathsf{tx}}, \mathsf{tx}, \mathsf{sl})$ we obtain the following:

$$
\hat{\mathbf{P}} = \begin{pmatrix}
0 & \frac{B}{4} & \frac{B}{2} & 0 & 0 & 0 \\
0 & 0 & 0 & 2(1-p) & 2p & 0 \\
0 & 0 & 0 & \frac{4(1-p)}{B} & \frac{4p}{B} & 0 \\
0 & \frac{B}{8} & \frac{B}{4} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{S}{2} \\
\frac{1}{S} & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
$$

For example, to compute the cell $\hat{\mathbf{P}}_{\mathsf{ss},\mathsf{bk}_1}$, we first obtain $\hat{\nu}_{\min}(\mathsf{ss}) = 1$, $\hat{\nu}_{\max}(\mathsf{bk}_1) = \frac{B}{2}$ and that the only possible transition from ss to bk_1 is labeled with the abstract scenario word $\boldsymbol{\mu}_6^1$ which has probability $\frac{1}{2}$. After that, we apply Definition 5.5 and we get:

$$
\begin{aligned}
\hat{\mathbf{P}}_{\mathsf{ss},\mathsf{bk}_1} &= \frac{\hat{\nu}_{\max}(\mathsf{bk}_1)}{\hat{\nu}_{\min}(\mathsf{ss})} \sum_{\mathsf{ss} \xrightarrow{\omega^\sharp} \mathsf{bk}_1 \in \Delta} \hat{\Pr}(\omega^\sharp) \\
&= \frac{B}{2} . \frac{1}{2}
\end{aligned}
$$

Input : Inequalities $I_0 = \{c_j + \sum_{1 \leq i \leq n} a_{i,j} x_i \leq 0 \mid 1 \leq j \leq m\}$
Input : Constraints C_0
Output : Set $\mathcal{I}$ of inequalities-constraints tuples $\langle C, I \rangle$

$\mathcal{I} \leftarrow \{\langle C_0, I_0 \rangle\}$;
for $i = 1$ **to** $n - 1$ **do**
  ⟅ *Eliminate unknown* x_i ⟆
  $\mathcal{I}' \leftarrow \varnothing$;
  foreach $\langle C, I \rangle \in \mathcal{I}$ **do**
    ⟅ *Decompose* I *depending on the sign of the coefficient of* x_i ⟆
    $I^+ \leftarrow \{\langle c_j + \sum_{i \leq k \leq n} a_{k,j} x_k \leq 0 \rangle \in I \mid a_{i,j} > 0\}$;
    $I^- \leftarrow \{\langle c_j + \sum_{i \leq k \leq n} a_{k,j} x_k \leq 0 \rangle \in I \mid a_{i,j} < 0\}$;
    $I^0 \leftarrow \{\langle c_j + \sum_{i \leq k \leq n} a_{k,j} x_k \leq 0 \rangle \in I \mid a_{i,j} = 0\}$;
    $I^? \leftarrow I \setminus (I^+ \cup I^- \cup I^0)$;
    ⟅ *For each possible sign combination in* $I^?$*, we generate a new case* ⟆
    $P \leftarrow \{\langle I^+, I^-, I^0, C \rangle\}$;
    foreach $\langle c_j + \sum_{i \leq k \leq n} a_{k,j} x_k \leq 0 \rangle \in I^?$ **do**
      $x \leftarrow \langle c_j + \sum_{i \leq k \leq n} a_{k,j} x_k \leq 0 \rangle$;
      $P \leftarrow \{\langle I^+ \cup \{x\}, I^-, I^0, C \wedge a_{i,j} > 0 \rangle \mid \langle I^+, I^-, I^0, C \rangle \in P\} \cup$
        $\{\langle I^+, I^- \cup \{x\}, I^0, C \wedge a_{i,j} < 0 \rangle \mid \langle I^+, I^-, I^0, C \rangle \in P\} \cup$
        $\{\langle I^+, I^-, I^0 \cup \{x\}, C \wedge a_{i,j} = 0 \rangle \mid \langle I^+, I^-, I^0, C \rangle \in P\}$;
    end
    ⟅ *We can apply now the classical Fourier-Motzkin elimination on each case* ⟆
    foreach $\langle I^+, I^-, I^0, C \rangle \in P$ **do**
      $I \leftarrow I^0$;
      foreach $\langle c^+ + \sum_{i \leq k \leq n} a_k^+ x_k \leq 0 \rangle \in I^+$ **do**
        foreach $\langle c^- + \sum_{i \leq k \leq n} a_k^- x_k \leq 0 \rangle \in I^-$ **do**
          $I \leftarrow I \cup \{\langle (c^- a_i^+ - c^+ a_i^-) + \sum_{i+1 \leq k \leq n} (a_k^- a_i^+ - a_k^+ a_i^-) x_k \leq 0 \rangle\}$;
        end
      end
      $\mathcal{I}' \leftarrow \mathcal{I}' \cup \{\langle C, I \rangle\}$;
    end
  end
  $\mathcal{I} \leftarrow \mathcal{I}'$;
end

Figure 5.9: Parametric Fourier-Motzkin elimination

At the end, to obtain our system (5.1) we just apply Theorem 5.1 by computing $\hat{\boldsymbol{\pi}} \leq \hat{\boldsymbol{\pi}}\mathbf{P}$ with the additional stochastic normalization condition $\boldsymbol{\pi}_{\mathsf{ss}} + \boldsymbol{\pi}_{\mathsf{bk}_1} + \boldsymbol{\pi}_{\mathsf{bk}_2} + \boldsymbol{\pi}_{\overline{\mathsf{tx}}} + \boldsymbol{\pi}_{\mathsf{tx}} + \boldsymbol{\pi}_{\mathsf{sl}} = 1$. ○

5.5 Resolution

Assume that we are interested in computing a proportional expectation η. Without loss of generality, assume also that η is semantically equivalent to the stationary probability of an abstract state $\sigma_\eta^\sharp$. To compute a safe range of $\hat{\boldsymbol{\pi}}_{\sigma_\eta^\sharp}$, we just have to perform a projection of the linear system $\hat{\boldsymbol{\pi}} \leq \hat{\boldsymbol{\pi}}\hat{\mathbf{P}}$ that keeps only the unknown $\hat{\boldsymbol{\pi}}_{\sigma_\eta^\sharp}$ and removes all the other unknowns while preserving all constraints. Many off-the-shelf symbolic environments provide such feature, however in practice, we have noticed that existing

methods do not scale well.

We propose to employ another approach using the Fourier-Motzkin projection algorithm. While most works employ this method on linear inequalities having real coefficients, we extend it to a parametric setting as in [Sur16]. We give in Figure 5.9 the details of the parametric version of the algorithm. We have as inputs a set I_0 of m parametric inequalities of the form $\{c_j + \sum_{1\leq i\leq n} a_{i,j}x_i \leq 0 \mid 1 \leq j \leq m\}$ where each x_i is an unknown and each c_j and $a_{i,j}$ are parametric coefficients of arbitrary form. Additionally, we also provide a (possibly empty) set of constraints C_0 that gives initial information about the parameters (for example, a parameter p_l of a Bernoulli distribution is always in the range $[0, 1]$). The aim of the algorithm is to return a set of constraints equivalent to I_0 where all variables are eliminated except a single one (assume x_n). To do so, the algorithm eliminates the other variables sequentially. At each iteration, a variable is eliminated and we obtain a set of parametric solutions $\{\langle C, I\rangle\}$ where I are a set of linear constraints on the remaining unknowns and C are the conditions on the parameters for obtaining the solution I.

For the sake of clarity, let us first describe the classical non-parametric version of the algorithm. To eliminate a variable x_i, we examine its coefficients $a_{i,j}$ in I and we partition the inequalities depending on the sign of these coefficients (lines 5 – 8). The idea is that when having two inequalities $\langle c_{j_1} + \sum_{i\leq k\leq n} a_{k,j_1}x_k \leq 0\rangle$ and $\langle c_{j_2} + \sum_{i\leq k\leq n} a_{k,j_2}x_k \leq 0\rangle$ where the signs of the target coefficients a_{i,j_1} and a_{i,j_2} are different and not null (let say $a_{i,j_1} > 0$ and $a_{i,j_2} < 0$), we can generate a new inequality $\langle a_{i,j_1}c_{j_2} - a_{i,j_2}c_{j_1} + \sum_{i\leq k\leq n}(a_{i,j_1}a_{k,j_2} - a_{i,j_2}a_{k,j_1})x_k \leq 0\rangle$ that is equivalent to the previous inequalities and where the coefficient of x_i is null. So, if we perform the same merging operation on every couple of sign-opposite inequalities, while keeping the inequalities that have already a null coefficient on x_i, we obtain a set of inequalities equivalent to the previous ones and where x_i has been eliminated (lines 14 – 19).

Example 5.6. We illustrate this process on our motivating example and its linear system (5.1). To compute the throughput proportional expectation θ we project on $\hat{\boldsymbol{\pi}}_{\mathsf{tx}}$ and we eliminate all other unknowns. A preliminary step is however necessary that gets rid of the normalization equality constraint, so we will manipulate later only inequalities. This is done by replacing every occurrence of $\hat{\boldsymbol{\pi}}_{\mathsf{bk}_1}$ for example with $1 - (\hat{\boldsymbol{\pi}}_{\mathsf{ss}} + \hat{\boldsymbol{\pi}}_{\mathsf{bk}_2} + \hat{\boldsymbol{\pi}}_{\mathsf{sl}} + \hat{\boldsymbol{\pi}}_{\mathsf{tx}} + \hat{\boldsymbol{\pi}}_{\overline{\mathsf{tx}}})$ and we obtain:

$$I = \begin{cases} 1 : \hat{\boldsymbol{\pi}}_{\mathsf{ss}} - \frac{1}{S}\hat{\boldsymbol{\pi}}_{\mathsf{sl}} \leq 0 \\ 2 : \left(-\frac{B}{8} - 1\right)\hat{\boldsymbol{\pi}}_{\overline{\mathsf{tx}}} + \left(-\frac{B}{4} - 1\right)\hat{\boldsymbol{\pi}}_{\mathsf{ss}} - \hat{\boldsymbol{\pi}}_{\mathsf{bk}_2} - \hat{\boldsymbol{\pi}}_{\mathsf{sl}} - \hat{\boldsymbol{\pi}}_{\mathsf{tx}} + 1 \leq 0 \\ 3 : -\frac{B}{4}\hat{\boldsymbol{\pi}}_{\overline{\mathsf{tx}}} - \frac{B}{2}\hat{\boldsymbol{\pi}}_{\mathsf{ss}} + \hat{\boldsymbol{\pi}}_{\mathsf{bk}_2} \leq 0 \\ 4 : 2p\hat{\boldsymbol{\pi}}_{\overline{\mathsf{tx}}} + \left(2p - \frac{4p}{B}\right)\hat{\boldsymbol{\pi}}_{\mathsf{bk}_2} + 2p\hat{\boldsymbol{\pi}}_{\mathsf{sl}} + 2p\hat{\boldsymbol{\pi}}_{\mathsf{ss}} + (2p+1)\hat{\boldsymbol{\pi}}_{\mathsf{tx}} - 2p \leq 0 \\ 5 : (2(1-p)+1)\hat{\boldsymbol{\pi}}_{\overline{\mathsf{tx}}} + \left(2(1-p) - \frac{4(1-p)}{B}\right)\hat{\boldsymbol{\pi}}_{\mathsf{bk}_2} + 2(1-p)\hat{\boldsymbol{\pi}}_{\mathsf{sl}} + 2(1-p)\hat{\boldsymbol{\pi}}_{\mathsf{ss}} + 2(1-p)\hat{\boldsymbol{\pi}}_{\mathsf{tx}} - 2(1-p) \leq 0 \\ 6 : \hat{\boldsymbol{\pi}}_{\mathsf{sl}} - \frac{S}{2}\hat{\boldsymbol{\pi}}_{\mathsf{tx}} \leq 0 \end{cases}$$

Now to eliminate $\hat{\boldsymbol{\pi}}_{\mathsf{sl}}$ we check its coefficients in every inequality, which are in order $\langle -\frac{1}{S}, -1, 0, 2p, 2(1-p), 1\rangle$. If we assume that $S > 0$ and $p \in]0,1[$, all these coefficients have fixed signs. Consequently, we can apply the non-parametric Fourier-Motzkin elimination directly. For example, the second and the fourth inequalities, that we denote by I_2 and I_4 respectively, have opposite signs coefficients in $\hat{\boldsymbol{\pi}}_{\mathsf{sl}}$, so we can combine them by computing:

$$(2p) \times I_2 - (-1) \times I_4 : -\frac{Bp}{4}\hat{\boldsymbol{\pi}}_{\overline{\mathsf{tx}}} - \frac{4p}{B}\hat{\boldsymbol{\pi}}_{\mathsf{bk}_2} - \frac{Bp}{2}\hat{\boldsymbol{\pi}}_{\mathsf{ss}} + \hat{\boldsymbol{\pi}}_{\mathsf{tx}} \leq 0$$

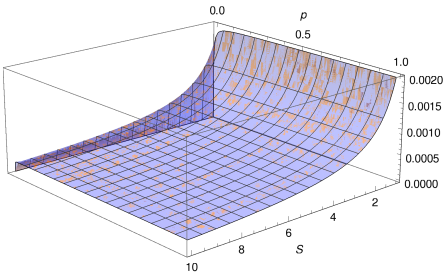
We can see that $\hat{\boldsymbol{\pi}}_{\mathsf{sl}}$ has been eliminated. By repeating this process for every couple of opposite-sign inequalities, all occurrences of the unknowns are eliminated. ○

When the coefficients of the unknowns are not constant, we can not always determine their signs. Consequently, we collect the set of undetermined inequalities $I^?$ (line 9) and we eliminate the ambiguity by pushing the sign conditions into the parameters constraints C. In other words, we fork the inequalities I into a set of new inequalities : one for every possible sign combination of the undertermined coefficients. For each case, the conditions of the sign combination are accumulated with the current parameters conditions C, and the undetermined inequalities are classified depending on these sign conditions (line 12). Therefore, all coefficient signs are resolved and the classical Fourier-Motzkin elimination can be applied.

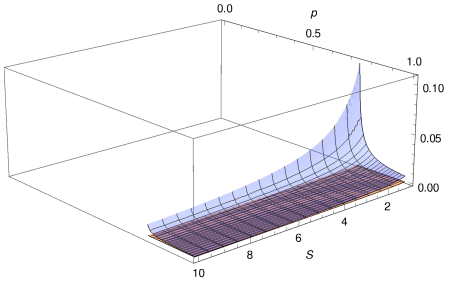
5.6 Experiments

The proposed approach has been implemented in a prototype analyzer called MARCHAL (*MARkov CHains AnaLyzer*) using the OCaml language, the CIL frontend and the Apron library. Also, we have implemented the parametric Fourier-Motzkin elimination algorithm in Mathematica. For our benchmarks, we have considered the simple sensing application described in our motivating example. We have varied the underlying communication scheme in order to compare the achieved throughput. The first case is URB (Unbounded Retransmissions with Backoff) which has been presented previously as the motivating example. The second communication scheme case is UR (Unbounded Retransmissions) that keeps sending a packet continuously until receiving an acknowledgment without any backoff. For both cases, we assume that the wireless link quality follows a Bernoulli distribution with parameter p and that an inter-sensing sleep period is determined by the parameter S. In addition, we assume that the time unit of the concrete chains is equal to $1ms$ and that the operations of sensing, receiving and transmitting without acknowledgment consume one unit, while the transmission with acknowledgment consumes two units.

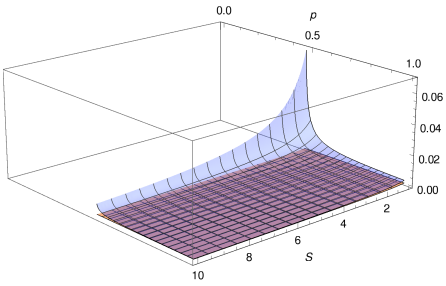
We show in Figure 5.10 the variation of the throughput θ as a function of the inter-sensing period S and the link quality p and the remaining parameter B has been



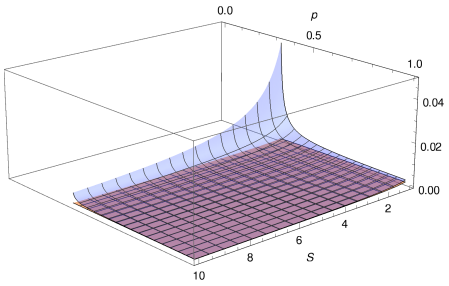
(a) UR



(b) URB, $U = 1$, $B = 4$

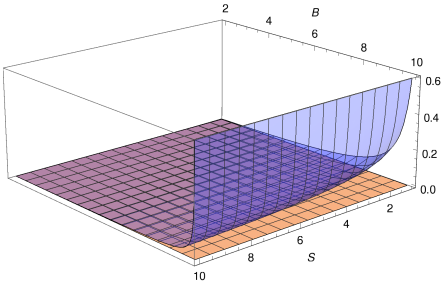


(c) URB, $U = 2$, $B = 4$

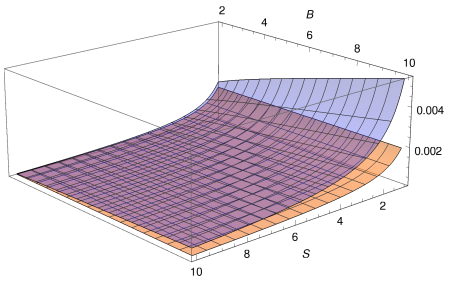


(d) URB, $U = 3$, $B = 4$

Figure 5.10: Variation of the throughput θ as a function of S and p.



(a) URB, $U = 1$



(b) URB, $U = 3$

Figure 5.11: Variation of the throughput θ as a function of S and B while fixing $p = 0.9$.

fixed to 4. For UR, we observe that the lower and upper bounds functions are the same, meaning that the exact analytic solution was found. For the case of URB, the analysis was less precise since we were able to get only *partial* results. Indeed, we notice that for some range of p, the analysis has not reported any bound. This is a natural shortcoming of over-approximation that will make the analysis loose some information during computations, which leads to potential imprecise inequalities on the stationary distribution. Nevertheless, we can see that the empty (unconstrained) region has been reduced by increasing the number of uniform partitions U.

Let us now discuss the variation of the throughput of URB[2] w.r.t. to S and B while fixing p to 0.9, which is shown in Figures 5.2(b) and 5.11. We can notice that

[2]UR is not shown since it does not depend on B.

Protocol	# States	# Parameters	Static analysis		PFM Resolution	
			Time (s)	Memory (MB)	Time (s)	Memory (MB)
UR	4	2	1	22	0.08	0.4
URB, $U = 1$	5	3	2	22	0.3	1
URB, $U = 2$	6	3	11	27	2	6
URB, $U = 3$	7	3	62	39	10	19

Table 5.1: Experiments benchmarks.

the precision of the analysis downgrades with the increase of the parameter B. This can be explained by the fact for high values of B, the fixed partitioning of `uniform` outcomes will merge into a single abstract state more incompatible states that are from sharing common properties, as the sojourn time for example. Since we use maximal and minimal sojourn times for approximating the steady state distribution, the gap between the concrete values induces a coarse over-approximation in the inequalities coefficients. Nevertheless, we notice that the precision increases with the increase of the uniform partitioning constant U.

Finally, we give in Table 5.1 the performance statistics of these experiments. We report for each case the size of the abstract Markov chain, the number of parameters, the time and memory consumption of the chain extraction phase and finally the time and memory consumption of the system resolution phase. In general, we notice an overall analysis time not exceeding two minutes with less 50 MB of peak memory consumption. It is worth noting that we have also tried to solve the inequalities system using the built-in Mathematica functions. However, Mathematica was not able to return within a 30mn timeout, except for the first case only.

5.7 Conclusion

We have presented a novel approach for obtaining guaranteed bounds of performance metrics of communication protocols. The method is based on the framework of abstract interpretation and proposes a novel Abstract Markov Chains domain for approximating the probabilistic semantics of programs. We have also explained how to exploit the information encapsulated within this domain in order to infer a sound approximation of the stationary distribution of the protocol, which is the key ingredient for computing a large range of performance metrics such as the throughput and the energy consumption. A prototype of the analysis have been presented along with some preliminary results.

Chapter 6

Conclusion & Perspectives

6.1 Conclusion

Developing reliable programs for complex and networked embedded systems is a difficult task. The presence of critical bugs or under-optimized resource policies can have negative effects on the proper operation of the network. Therefore, it is vital to discover these errors as early as possible during the development phase in order to avoid potential crashes when the system becomes operational.

In this thesis, we have proposed automatic verification approaches that can certify the reliability of an embedded program with respect to user-defined specifications. To do so, we have employed the theory of abstract interpretation which offers a rigorous formal framework for developing sound static analysis techniques covering the entire behaviors space in finite time. More particularly, we have employed this theory to target to important and very distinct aspects of program reliability.

The first aspect is qualitative and concerns the correctness of device drivers, which represents a crucial requirement for the reliability of wireless embedded networks. Drivers occupy a central and critical place within the software stack, since they are the solely responsible of low-level interactions with the different hardware peripherals of the system. Existing automatic verification tools for wireless embedded networks are not adapted to this particular type of errors and can not therefore certify the correctness of device drivers at the hardware interaction semantics layer. We have proposed a novel static analysis of device drivers that can certify the absence of erroneous hardware management policies in a TinyOS program. We proposed several abstract domains for tracking sensitive information about the device driver, the hardware and the interrupts. In addition, we embedded into the analysis a precise concurrency model that formalizes the preemption mechanism of interrupts. To scale the analysis to large

programs, we presented a compositional analysis that computes an over-approximation of the preemption contexts for each interrupt and injects their return contexts at firing locations. A prototype of our abstract interpreter has been implemented and tested on several ADPs for various hardware sub-systems in the ATmega128 MCU. These test cases covered many aspects of device drivers programming, such as active polling, interrupt-based serial transfer and I/O register configuration. For each ADP, we analyzed the respective device driver and we measured the total analysis time, the amount of consumed memory and the number of reported false alarms. In most cases, the analyzer was able to prove the property with acceptable performances and with high precision.

The second aspect of software reliability addressed in this thesis is of a quantitative nature and concerns the performance evaluation of wireless communication protocols. More precisely, we targeted a family of widely used performance indicators that express some velocity over time, such as the duty cycle and the packets throughput. Thanks to the soundness guarantee of abstract interpretation, our analysis is able to find safe bounds of these metrics that are valid for every possible execution. In addition, the obtained bounds are symbolic and expressed as functions of the parameters of the program. The proposed solution is based on a novel Abstract Markov Chain domain that can over-approximate families of possibly infinite Markov chains. This new structure is guaranteed to have finite size and is inferred by analyzing the source code of the program. We also presented a theoretical result that allows extracting a system of linear and parametric inequalities that can bound the stationary distribution of the Makrov chain associated to the program. To solve this system, we described a projection algorithm based on the Fourier-Motzkin elimination method that we adapted to a parametric setting. A prototype of the analysis has been implemented and some preliminary results has been discussed

6.2 Perspectives

We would like to extend our driver analysis tool in different ways. First, the current version is tailored to the ATmega128 MCU only and we plan to extend its support to other microcontroller families. We envisage to target the famous MSP430 16-bits MCU, and also the promising ARM Cortex M0 32-bits architecture implemented in various MCUs, such as Nordic nRF51 and STM32 F0 MCUs. To support these platforms, new ADPs should be formalized to express the specific hardware behaviors of their different subsystems. Also, we plan to extend the presented analysis to other execution models of wireless embedded networks. An interesting case is the *protothreads* paradigm [DSVA06] implemented in the Contiki operating system. This programming abstraction

is different from the task-driven execution model of TinyOS and allows developing programs in a thread-like style, which is more "natural" in the context of event-driven programs and has been proved to reduce their implementation complexity.

The quantitative analyzer is still in a preliminary phase and many features can be added. Firstly, we want to analyze real-world implementations of communication protocols that are part of existing OSs, such as TinyOS and Conitki. More particularly, we want to focus on the standard networking stack that is being developed by different IETF working groups, such as TSCH and RPL. Secondly, we have until now analyzed single-node scenarios and we would like to extend the approach for multi-nodes networks. The challenge being to scale the analysis to reasonable networks sizes, modular analysis seems the most appropriate solution. Finally, we have experienced in some situations imprecise approximations that lead to unbounded stationary distributions. We think that the primary reason is the fixed partitioning of `uniform` outcomes that may blindly merge incompatible transitions. We believe that more adequate abstractions for this particular distribution are necessary to enhance the precision of the analysis, especially for backoff mechanisms that are a very frequent pattern in communication protocols.

Bibliography

[Atm11] Atmel. Atmega128(l) datasheet. `www.atmel.com/images/doc2467.pdf`, 2011.

[BEFFH16] G. Barthe, T. Espitau, L. Ferrer Fioriti, and J. Hsu. Synthesizing probabilistic invariants via doob's decomposition. In *Proc. of the 28th International Conference on Computer Aided Verification (CAV '16)*, LNCS, pages 43–61. Springer, 2016.

[BGP+16] O. Bouissou, E. Goubault, S. Putot, A. Chakarov, and S. Sankaranarayanan. *Uncertainty Propagation Using Probabilistic Affine Forms and Concentration of Measure Inequalities*, volume 9636 of *LNCS*, pages 225–243. Springer, 2016.

[BISV08] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In *Proc. 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08)*, pages 43–56. ACM, 2008.

[BK11] D. Bucur and M. Kwiatkowska. On software verification for sensor nodes. *Journal of Systems and Software*, 84(10):1693–1707, 2011.

[BNS10] J. Brauer, T. Noll, and B. Schlich. Interval analysis of microcontroller code using abstract interpretation of hardware and software. In *Proc. 13th International Workshop on Software & Compilers for Embedded Systems (SCOPES '10)*, pages 1–10, 2010.

[Buc12] D. Bucur. Temporal monitors for TinyOS. In *Proc. 3rd International Conference on Runtime Verification (RV '12)*, volume 7687 of *LNCS*, pages 96–109. Springer, 2012.

[CAE+07] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proc. 5th International Conference on Embedded Networked Sensor Systems (SenSys '07)*, pages 205–218. ACM, 2007.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd International Symposium on Programming (ISOP '76)*, pages 106–130, 1976.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM, 1977.

[CC02] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *Proc. 11th International Conference on Compiler Construction (CC '11)*, volume 2304 of *LNCS*, pages 159–178. Springer, 2002.

[CDE08] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, 2008.

[CE82] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proc. Workshop on Logic of Programs*, pages 52–71. Springer-Verlag, 1982.

[CGMP99] E. M. Clarke, O. Grumberg, M. Minea, and D. A. Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer (STTT)*, 2(3):279–287, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th Symposium on Principles of Programming Languages (POPL '78)*, pages 84–97. ACM, 1978.

[CHA+07] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *Proc. 16th European Symposium on Programming (ESOP'07)*, volume 4421 of *LNCS*, pages 520–535. Springer, 2007.

[CKL04] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[Cou78] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes*. PhD thesis, Joseph Fourier University, Grenoble, 1978.

[Cou85] R. Cousot. *Fondements des méthodes de preuve d'invariance et de fatalité de programmes parallèles.* Thèse d'État ès sciences mathématiques, Institut National Polytechnique de Lorraine, Nancy, France, 1985.

[Cou99] P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design.* NATO ASI Series F. IOS Press, 1999.

[CR06] N. Cooprider and J. Regehr. Pluggable abstract domains for analyzing embedded software. In *Proc. ACM Conference on Language, Compilers, and Tool Support for Embedded Systems (LCTES '06)*, pages 44–53. ACM, 2006.

[CS13] A. Chakarov and S. Sankaranarayanan. Probabilistic program analysis with martingales. In *Proc. 25th International Conference on Computer Aided Verification (CAV '13)*, LNCS, pages 511–526. Springer, 2013.

[CS14] A. Chakarov and S. Sankaranarayanan. *Expectation Invariants for Probabilistic Program Loops as Fixed Points*, volume 8723 of *LNCS*, pages 85–100. Springer, 2014.

[DGV04] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. 29th IEEE International Conference on Local Computer Networks (LCN '04)*, pages 455–462, 2004.

[DR10] J. Duan and J. Regehr. Correctness proofs for device drivers in embedded systems. In *Proc. 5th International Workshop on Systems Software Verification (SSV'10)*, pages 5–5. USENIX Association, 2010.

[DSL14] F. Despaux, Y. Q. Song, and A. Lahmadi. Modelling and performance analysis of wireless sensor networks using process mining techniques: ContikiMAC use case. In *Proc. IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS '14)*, pages 225–232, 2014.

[DSVA06] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proc. 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*, pages 29–42. ACM, 2006.

[FAO03] FAO. Water management : Towards 2030. `http://www.fao.org/ag/magazine/0303sp1.htm`, 2003.

[FDSL15] F. F. Despaux, Y. Q. Song, and A. Lahmadi. Extracting markov chain models from protocol execution traces for end to end delay evaluation in

wireless sensor networks. In *Proc. IEEE World Conference on Factory Communication Systems (WFCS '15)*, pages 1–8, 2015.

[Fer01a] J. Feret. Abstract interpretation-based static analysis of mobile ambients. In *Proc. of the 8th International Symposium on Static Analysis (SAS '01)*, volume 2126 of *LNCS*, pages 412–430. Springer, 2001.

[Fer01b] J. Feret. Occurrence counting analysis for the pi-calculus. In *Workshop on GEometry and Topology in COncurrency theory*, volume 39.(2) of *Electronic Notes in Theoretical Computer Science*, 2001.

[Fox03] A. Fox. Formal specification and verification of arm6. In *Proc. 16th International Conference on Theorem Proving in Higher Order Logics (TPHOLs '03)*, volume 2758 of *LNCS*, pages 25–40. Springer, 2003.

[Fru06] M. Fruth. Probabilistic model checking of contention resolution in the IEEE 802.15.4 low-rate wireless personal area network protocol. In *Proc. 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '06)*, pages 290–297, 2006.

[GLvB+03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. Conference on Programming Language Design and Implementation (PLDI '03)*, pages 1–11. ACM, 2003.

[GMR+13] L. Gallina, A. Marin, S. Rossi, T. Han, and M. Kwiatkowska. A process algebraic framework for estimating the energy consumption in ad-hoc wireless sensor networks. In *Proc. 16th ACM International Conference on Modeling, Analysis & Simulation of Wireless and Mobile Systems (MSWiM '13)*, pages 255–262. ACM, 2013.

[HHZ11] E. Hahn, H. Hermanns, and L. Zhang. Probabilistic reachability for parametric markov models. *International Journal on Software Tools for Technology Transfer*, 13(1):3–19, 2011.

[HVL+10] W. Hedgecock, P. Völgyesi, A. Ledeczi, X. Koutsoukos, A. Aldroubi, A. Szalay, and A. Terzis. Mobile air pollution monitoring network. In *Proc. of the ACM Symposium on Applied Computing (SAC '10)*, pages 795–796. ACM, 2010.

[JM09] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proc. 21st International Conference on Computer Aided Verification (CAV '09)*, volume 5643 of *LNCS*, pages 661–667. Springer, 2009.

[KF94] M. Kaminski and N. Francez. Finite-memory automata. *Theoretical Computer Science*, 134(2):329 – 363, 1994.

[KLM+15] D. Kroening, L. Liang, T. Melham, P. Schrammel, and M. Tautschnig. Effective verification of low-level software with nested interrupts. In *Proc. Design, Automation & Test in Europe (DATE '15)*, pages 229–234, 2015.

[KNP11] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[LBV06] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proc. 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*, pages 174–174. IEEE Computer Society, 2006.

[LGJ07] T. Le Gall and B. Jeannet. Lattice automata: A representation for languages on infinite alphabets, and some applications to verification. In *Proc. of the 14th International Symposium on Static Analysis (SAS '07)*, LNCS, pages 52–68. Springer Berlin Heidelberg, 2007.

[LMP+04] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. Springer, 2004.

[LR10] P. Li and J. Regehr. T-Check: Bug finding for sensor networks. In *Proc. 9th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN '10)*, pages 174–185. ACM, 2010.

[Min04] A. Miné. *Weakly relational numerical abstract domains.* PhD thesis, École Polytechnique, Paris, 2004.

[Min06a] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '06)*, pages 54–63. ACM, 2006.

[Min06b] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation (HOSC)*, 19(1):31–100, 2006.

[Min11] A. Miné. Static analysis of run-time errors in embedded critical parallel C programs. In *Proc. 20th European Symposium on Programming (ESOP '11)*, volume 6602 of *LNCS*, pages 398–418. Springer, 2011.

[Min12] A. Miné. Abstract domains for bit-level machine integer and floating-point operations. In *Proc. 4th International Workshop on Invariant Generation (WING '12)*, page 16, 2012.

[Min14] A. Miné. Relational thread-modular static value analysis by abstract interpretation. In *Proc. 15th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI '14)*, volume 8318 of *LNCS*, pages 39–58. Springer, 2014.

[MMHS11] P. Mardziel, S. Magill, M. Hicks, and M. Srivatsa. Dynamic enforcement of knowledge-based security policies. In *Proc. IEEE 24th Computer Security Foundations Symposium (CSF '11)*, pages 114–128, 2011.

[Mon07] D. Monniaux. Verification of device drivers and intelligent controllers: A case study. In *Proc. 7th International Conference on Embedded Software (EMSOFT '07)*, pages 30–36, 2007.

[MVO+10] L. Mottola, T. Voigt, F. Österlind, J. Eriksson, L. Baresi, and C. Ghezzi. Anquiro: Enabling efficient static verification of sensor network software. In *Proc. Workshop on Software Engineering for Sensor Network Applications (SESENA '10)*, pages 32–37. ACM, 2010.

[NMRW02] G. Necula, S. McPeak, S. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Proc. 11th International Conference on Compiler Construction (CC '02)*, pages 213–228, 2002.

[OT03] V. Orgovan and M. Tricker. An introduction to driver quality. Microsoft WinHec 2004 Presentation DDT301, 2003.

[Ox12] Happy Ox. Issue #149: AVR overflow in async timer. `https://github.com/tinyos/tinyos-main/issues/149`, 2012.

[Par66] R. Parikh. On context-free languages. *Journal of ACM*, 13(4):570–581, 1966.

[PE08] F. J. Pierce and T. V. Elliott. Regional and on-farm wireless sensor networks for agricultural systems in eastern washington. *Comput. Electron. Agric.*, 61(1):32–43, 2008.

[PPH12] J. A. Propst, K. M. Poole, and J. O. Hallstrom. An embedded sensing approach to monitoring parking lot occupancy. In *Proc. 50th Annual Southeast Regional Conference (ACM-SE '12)*, pages 309–314, 2012.

[PRP+06] J. Panchard, S. Rao, T. Prabhakar, H. S. Jamadagni, and J. Hubaux. COMMON-Sense Net: Improved water management for resource-poor farmers via sensor networks. In *Proc. International Conference on Information and Communication Technologies and Development (ICTD '06)*, pages 22–33, 2006.

[QS82] J. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proc. 5th International Symposium on Programming (ESOP '82)*, volume 137 of *LNCS*, pages 337–351. Springer, 1982.

[RDH03] Robby, Matthew B. D., and J. Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *Proc. 9th European Software Engineering Conference (ESEC '03)*, pages 267–276. ACM, 2003.

[Reg05] J. Regehr. Random testing of interrupt-driven software. In *Proc. 5th ACM International Conference on Embedded Software (EMSOFT '05)*, pages 290–298. ACM, 2005.

[RRW03] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. In *Proc. Third International Conference on Embedded Software (EMSOFT '03)*, volume 2855 of *LNCS*, pages 306–322. Springer, 2003.

[SCG13] S. Sankaranarayanan, A. Chakarov, and S. Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *Proc. 34th ACM Conference on Programming Language Design and Implementation (PLDI '13)*, pages 447–458. ACM, 2013.

[SLA+10] R. Sasnauskas, O. Landsiedel, M. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: Discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proc. 9th International Conference on Information Processing in Sensor Networks (IPSN '10)*, pages 186–196. ACM, 2010.

[Sur16] P. Suriana. Fourier-Motzkin with non-linear symbolic constant coefficients. Master thesis, Massachusetts Institute of Technology, 2016.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5(2):285–309, 1955.

[Vil02] S. Villemot. Automates finis et intérpretation abstraite: Application à l'analyse statique de protocoles de communication. Rapport de DEA, École normale supérieure, 2002.

[WALJ+06] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 381–396. USENIX Association, 2006.

[WHWX07] Y. Wang, L. Huang, J. Wu, and H. Xu. Wireless sensor networks for intensive irrigated agriculture. In *Proc. 4th IEEE Consumer Communications and Networking Conference (CCNC '07)*, pages 197–201, 2007.