Formal Analysis of Communication Protocols for Wireless Sensor Systems

Ivaylo Valkov

Submitted in fulfilment of the requirements for the
Degree of Doctor of Philosophy

School of Computing Science
College of Science and Engineering
University of Glasgow

October 2023

Abstract

Sensor technology is an increasingly popular area of research due to the prevalent use of sensor devices. With the need for accurate, detailed data sensors are increasingly often used together in sensor networks. As the size of these sensor networks grows, so does the importance of efficient methods for their analysis for the prevention of system errors and discovery of design flaws. The increasing number of sensor devices leads to an exponential increase is the state space of the associated model. As such models of realistic systems are decreasingly often small enough for their verification to be feasible. Symmetry reduction techniques developed over the last 30 years, have been shown to be effective in reducing the state space explosion problem, particularly in the case of heterogeneous sensor systems, which contain many identical sensor devices.

In this thesis we present our approach to verifying Ctrl-MAC, a novel wireless network protocol that supports bidirectional communication of multiple simultaneous physical properties. We explore the extent to which symmetry reduction can aid the model checking process for a sensor network communication protocol. We present our results, and suggest statistical approaches based on our observations of the protocol.

We investigate the use of automated tools for the application of symmetry reduction, in particular GRIP, which is well suited for symmetry reduction of wireless sensor network systems. Models of communication protocols often require the use of synchronisation to model the interaction between devices. We present GRIP 3.0, a new version of the tool, which provides support for the use of synchronised transition statements. We provide results from practical work, coupled together with a discussion of drawbacks and future improvements.

Contents

List of Tables

List of Figures

Acknowledgements

First and foremost, Prof. Alice Miller for all of the effort and patience that was needed in supervising my Ph.D. I deeply appreciate you going above and beyond in your duties and always making it feel like my supervision is of top importance. Every chapter of this thesis has been guided by Alice's indispensable advice and ideas, has been thoroughly proof-read by her, and substantially improved by her comments. I would not have gotten to the end if I was supervised by anybody else. Thank you, Alice!

I would like to thank my second supervisor Dr Michele Sevegnani and everybody else from the Science of Sensor Systems Software (S4) programme grant for the welcome introduction to the world of research. I am also grateful to S4, and in particular to Prof. Muffy Calder, for funding my Ph.D.

A big thank you to Prof. Alastair Donaldson for lengthy discussions about the GRIP tool and invaluable insight into its implementation.

Also thank you to Dr Simon Rogers for useful discussions which led to our proposed statistical approach.

I would like to thank Dr Gethin Norman and Dr Ornela Dardha, my annual progress review examiners. You helped me steer my thesis into the correct path.

Thank you to all of my friends for all of the help, advice, laughs, talks and so much more. All of you made my life better in your own way during my the last five years. Special thanks to Dr Alex Pancheva, who started her Ph.D journey at the same time as me, but has long since graduated and is my guiding light.

Thank you to my fellow PhD students from room G161. Michael, Frances, Ben, Sofiat, Doug, Peace, although we spent a large amount of time working from home during the crazy times of Covid19, I appreciate all of the days we shared working together.

A special thank you to my girlfriend Ari, who made the final push in the writing of this thesis so much more manageable.

A final huge thank you to my family, who have always been there for me, and I know will always be there for me. Mum and Dad, I cannot imagine better parents than you. My little sister Geri, you are without a doubt a better version of me. My grandpa Gancho, who sadly passed away before I could graduate, and my grandmas Ivanka and Todorka, I hope I can continue making you proud of me. I love you all!

Chapter 1

Introduction

1.1 Motivation

Sensor systems play an important role in everyday life. From the moment we wake up, we interact with sensor-driven technologies. In our homes, sensors monitor temperature and motion, so that appropriate heating and lighting is provided, and smart appliances employ various sensors to enhance convenience and energy efficiency. As we step outside, our smartphones act as multifunctional sensor hubs, collecting data from GPS, accelerometers, and ambient light sensors to provide location-based services, fitness tracking, and adaptive screen brightness. In transportation, sensor systems enable advanced driver-assistance features, such as parking sensors, lane departure warnings, and collision avoidance systems. Sensor networks monitor traffic flow, optimise logistics, and enhance public safety in smart cities. In healthcare, wearable sensors track vital signs and activity levels, empowering individuals to monitor their well-being, while medical devices employ sophisticated sensors to diagnose, treat, and manage various conditions. Even in entertainment, virtual reality headsets rely on motion sensors to create immersive experiences. Sensor systems have truly permeated our lives, enhancing our comfort, safety, and efficiency while driving innovation across countless domains.

Several challenges are associated with sensor networks, which can hinder their effectiveness and reliability. One major challenge is the limited power supply available to individual sensors. Since many sensors are deployed in remote or inaccessible locations, replacing or recharging batteries can be impractical or even impossible. This constraint necessitates the development of energy-efficient sensors and protocols that optimise power consumption and extend the network's lifetime. Additionally, the deployment of sensor networks in harsh environments, such as extreme temperatures or high humidity, poses durability challenges. Sensors must be designed to withstand these conditions to ensure accurate and consistent data collection.

Another challenge lies in the communication and coordination among sensors within the network. As the number of sensors increases, the complexity of managing and routing data also grows. Efficient routing algorithms and protocols need to be implemented to ensure timely

and reliable data transmission while minimising network congestion and energy consumption [241]. Moreover, sensor networks may face issues related to network coverage and connectivity. Obstacles like buildings, terrain, or interference from other devices can disrupt or weaken the wireless signals, resulting in data loss or limited coverage areas.

The volume of data generated by sensor networks can also pose a significant challenge for data processing and analysis. Managing and extracting meaningful insights from massive amounts of sensor data in real-time can be complex and resource-intensive. Advanced data analysis techniques, such as machine learning and artificial intelligence, are employed to process, filter, and extract valuable information from the sensor data.

Formal methods are a set of mathematically-based techniques that are used for the verification of software and hardware systems in order to ensure their correctness and reliability. They play a significant role in addressing the challenges faced by sensor networks by providing rigorous and systematic approaches to design, analysis, and verification.

Firstly, formal methods help optimise the energy consumption of sensor networks by enabling rigorous modelling and analysis of energy-related aspects. Through mathematical modelling and formal verification techniques, energy-efficient protocols, algorithms, and power management strategies can be designed and verified to maximise the network's lifetime and minimise energy consumption [21, 131].

Secondly, formal methods aid in designing efficient routing algorithms and protocols for sensor networks. They enable formal modelling and analysis of communication protocols, ensuring properties such as reliability, scalability, and timeliness. Formal verification techniques help identify and quantify potential issues, such as data loss or congestion, and verify the correctness of routing schemes, improving the network's overall performance and coordination. Furthermore, they can assist in analysing and optimising network coverage and connectivity. By employing formal modelling techniques, coverage problems can be detected, and deployment strategies can be optimised to ensure sufficient coverage and connectivity [85, 151, 210]. Formal verification can help identify areas with weak connectivity or potential signal interference, leading to the development of solutions to mitigate these issues. In particular, in data processing and analysis in sensor networks, formal models and algorithms can be developed to extract valuable insights from sensor data, allowing for efficient and accurate analysis. Formal verification techniques can ensure the correctness of data processing algorithms and the accuracy of the obtained results.

By leveraging formal methods, sensor networks can benefit from rigorous analysis, verification, and optimisation techniques. This leads to the development of more reliable, efficient, and secure sensor systems, enabling their successful deployment in diverse domains and addressing the challenges associated with energy consumption, communication, connectivity, security, and data processing.

Cyber-Physical Systems (CPSs) refer to integrated systems that combine physical compo-

nents with computational and networking elements. These systems bridge the gap between the physical and virtual worlds by connecting and coordinating physical processes with digital information processing, communication, and control. They typically involve a network of sensors, actuators, embedded systems, and computing devices that interact with the physical world and exchange data and information through wired or wireless communication networks. These systems are designed to monitor, control, and optimise various processes and environments, ranging from transportation and manufacturing to healthcare and energy.

The core aspect of CPSs is the integration of physical processes and objects with computational systems and networks. Physical components can include machinery, devices, vehicles, infrastructure, and even humans, while cyber components involve software, algorithms, data, and communication networks. CPSs enable real-time interactions between the physical and virtual components. They continuously sense the physical state, collect data, process information, and actuate physical processes accordingly. This real-time aspect is crucial for achieving responsive and dynamic control. This type of systems typically employ a wide range of sensors to collect data about the physical environment, such as temperature, pressure, motion, or location. Actuators are used to control physical processes by manipulating the physical elements, such as turning on or off a motor or adjusting a valve. Communication networks enable the exchange of data and information between physical and cyber components of CPSs. This is necessary for coordination, synchronisation, and collaboration among different components of the system.

Applications of CPSs can be found in various domains, including:

- Transportation: Intelligent transportation systems, autonomous vehicles, traffic control, and monitoring [216].
- Manufacturing: Smart factories, industrial automation, robotics, and supply chain management [238].
- Healthcare: Remote patient monitoring, medical device integration, telemedicine, and personalised healthcare [160].
- Energy: Smart grids, energy management, renewable energy integration, and demand response systems [136].
- Infrastructure: Smart buildings, smart cities, infrastructure monitoring, and crowd flow optimisation [160].

Wide Area Cyber-Physical Systems (WA-CPSs) refer to a specific subclass of Cyber-Physical Systems that operate over large geographical areas. WA-CPSs extend the capabilities of traditional CPSs by enabling coordination and control over a wide geographic scope, often involving diverse physical processes and heterogeneous components. They encompass a wide area, often including multiple physical locations or regions. These systems are designed

to monitor, control, and optimise physical processes and resources that are geographically dispersed.

WA-CPSs consist of interconnected subsystems distributed across different locations. These subsystems can be independent CPS or smaller-scale systems that are integrated to form a larger-scale system. The interconnection allows for information exchange and collaboration among different components. Typically, they involve diverse physical processes, components, and technologies. These systems may incorporate various types of sensors, actuators, communication networks, and computational elements. Managing the heterogeneity and ensuring interoperability among different subsystems can be a significant challenge.

Communication and networking play a crucial role in WA-CPSs to enable coordination and information exchange over a wide area. These systems rely on robust and reliable communication infrastructure to transmit data and commands among different subsystems. It is often necessary to design WA-CPSs with scalability and resilience in mind. The systems should be capable of handling large-scale operations and be resilient to failures, disruptions, or cyber attacks. Redundancy, fault tolerance, and backup mechanisms are important considerations in ensuring system reliability.

Applications of Wide Area Cyber-Physical Systems can be found in various domains, including:

- Smart Grids: Wide area monitoring and control of power generation, transmission, and distribution systems [208].
- Environmental Monitoring: Large-scale monitoring and management of environmental parameters like air quality, water resources, and weather conditions [203].
- Transportation Networks: Wide area traffic management and optimisation, including traffic flow control and congestion mitigation [74].
- Disaster Management: Coordination of emergency response systems across multiple regions during natural disasters or critical events [72].
- Supply Chain Management: Integrated monitoring and optimisation of supply chains across different locations [223].

WA-CPSs involve actuators, controllers and low-powered sensors, and are used in areas which span multiple kilometres. Such systems require a bounded communication delay in order to work properly; however, current wireless communication technologies are unable to provide bounded delay on wide areas [22]. *Ctrl-MAC* is a Low-Power Wide Area (LPWA) MAC protocol, proposed as a solution for WA-CPSs, and has been developed by partners in the Science of Sensor Systems Software (S4) project. Ctrl-MAC has been evaluated against LoRaWAN, a

state-of-the-art LPWA MAC protocol, by testing both on a varied number of nodes and different transmission patterns [22]. Results show that Ctrl-MAC performs better than LoRaWAN in terms of the communication requirements of reliable communication, two-way communication and bounded delays. The delay analysis performed, however, relies on numerical simulation for varying data loads. However, such simulations, by their nature, cannot analyse all possible network behaviours. To obtain a proof for the boundedness of the delay, we turn to formal methods.

The problem of proving that a bound for the communication delay of a network protocol exists and finding the value of that bound is a fundamental question of communication technologies. Ctrl-MAC is a protocol in which the communication delay is closely tied to the number of participating devices. This presents an issue for the typical modelling approach of creating a model for a small number of devices and extrapolating results for a larger number of devices. In this thesis we create a sequence of formal models for Ctrl-MAC with increasing levels of abstraction to generate results for small to moderately sized systems. We then use the knowledge gained from developing these models to produce a set of generic mathematical results and modelling techniques which can be applied to provide a bound on the transmission delay for sensor systems of any given size. Although our approach focuses on Ctrl-MAC, we anticipate that it can also be applied to other similar protocols.

1.2 Contributions

1.2.1 Thesis Statement

Thesis title: *Formal Analysis of Communication Protocols for Wireless Sensor Systems*.

Thesis statement: Formal methods can be used to provide guarantees of correctness for sensor network communication protocols for systems of moderate size, and to aid in the development of generic mathematical results for much larger systems. We demonstrate this by presenting a suite of PRISM models, properties and verification results for the Ctrl-MAC protocol, together with theoretical results, which will be used to prove correctness for systems of any given size.

I was funded by the EPSRC Science of Sensor Systems Software programme grant (EP/N007565/1, 2016-2022), and my work has been influenced by conversations and discussions with researchers from the project. Whilst writing up this thesis I was partially supported by the UKRI Strategic Priorities Fund to the UKRI Research Node on Trustworthy Autonomous Systems Governance and Regulation (EP/V026607/1, 2020-2024).

1.2.2 Main Results

We have developed a series of models for the Ctrl-MAC protocol. These were developed iteratively as a result of a series of conversations with the protocol developers and were used to establish a number of issues with the protocol specification. A sensor communication protocol such as Ctrl-MAC where the performance of protocol is dependant on the recent history of the system is what led us to search for new modelling techniques.

We considered a number of different techniques for dealing with the state space explosion encountered by our models and focused our efforts on symmetry reduction through the use of counter abstraction. We successfully used this technique to create two new models of Ctrl-MAC which employed different levels of counter abstraction and could be used to verify different scopes of properties. Doing so we obtained a deeper understanding of how Ctrl-MAC functions, and used that knowledge to propose a statistical approach that could be used to provide an approximation for the behaviour of the protocol.

Throughout our survey into symmetry reduction techniques as a solution to the state space explosion challenges faced by Ctrl-MAC, we looked into GRIP, one of the two tools for performing symmetry reduction in PRISM. This tool was the one that was more suited to the type of systems that Ctrl-MAC is (i.e. a large number of less complex devices). We have performed two sets of changes to GRIP. The first we have released as GRIP 2.1 and that one restores GRIP's functionality to that of its last release of about ten years ago. The second set of changes allows GRIP to support models using synchronised actions, the main obstacle in applying GRIP to Ctrl-MAC. We present this new feature as GRIP 3.0, a new version of the symmetry reduction tool. We applied GRIP 3.0 to a variety of PRISM models involving synchronised commands, achieving mixed results based on the extent of the model's use of synchronised statements.

1.3 Organisation of Thesis

Chapter 1 introduces the topics of formal methods and sensor systems, and provides an overview of how this thesis is structured. In Chapter 2 we provide some background to our area of research, giving a review of model checking and wireless sensor network literature. We present different model checking tools with a focus on the PRISM model checker. In addition, we summarise the main issues related to model checking and to wireless sensor networks, and the existing approaches to solving them. In this Chapter we also give an in-depth review of the symmetry reduction technique and its use in addressing the state space explosion problem. In Chapters 3 and 4, we describe the key technical concepts used in this thesis, related to model checking, and to Ctrl-MAC, respectively.

In Chapter 5 we present the initial PRISM models that we created during our formal analysis of Ctrl-MAC. We provide comparisons of the models and the results we obtain from them. Lastly, we present a statistical approach that could be used to obtain verification results for Ctrl-

MAC instances with a larger number of participating sensor devices. In chapter 6 we introduce the GRIP symmetry reduction tool. We then present our contributions to the tool: GRIP 2.1 and GRIP 3.0; as well as all of the changes that we performed to introduce support for synchronised statements to GRIP. We outline the extensions of the SPSL translation rules necessary for GRIP to support counter abstraction of PRISM specifications containing synchronised commands.

Lastly, in Chapter 7 we give a summary of the thesis and our results. We outline the challenges we encountered, our key contributions, and propose extensions to our contribution that could be made in the future. In the end, we list two appendices, in which we list some of the PRISM specifications referred to in the thesis, together with a list of abbreviations used.

Chapter 2

Background

In this chapter we highlight and present research relevant to this thesis. We discuss formal verification techniques, focusing mainly on model checking. We present different model checking tools, providing more detail about the PRISM model checker, which is the main model checker we use in this thesis (and is discussed in further detail in Section 3.5.1). We introduce a common problem with model checking, namely the state space explosion problem, and survey methods used to resolve it, including symmetry reduction which we exploit later in the thesis. We introduce wireless sensor networks, the problems frequently faced by them, and associated related verification approaches.

2.1 Formal Methods

The term "formal methods" is used to describe various mathematical techniques that are used in the formal specification and analysis of hardware and software systems. They are based on a formal specification language in combination with a collection of tools, which are used for checking specifications written in the language, as well as proving properties of that specification. Formal methods enable the verification of a system independently of its implementation.

Failures of software and hardware systems can result in significant financial losses and, in safety-critical contexts, even lead to fatal outcomes [135, 159, 180]. It is of great importance to be able to ensure the correctness of such systems. Consequently, there has been a growing focus on utilising formal verification to create high-confidence systems [102, 179].

Formal verification is the application of rigorous mathematics-based techniques with the purpose of proving or disproving the correctness of a computer-based system. The system, either software or hardware [140], is checked by ascertaining whether or not its behaviour satisfies a desired *property*. First, these properties need to be specified in order to define what the desired behaviour of the system is. Then a suitable formal model of the system that reflects the behaviour of the system needs to be built. This model must capture all aspects of the system that influence the behaviour of the system to ensure that the obtained correctness results are meaningful. At

the same time the model should be sufficiently abstract to make it understandable and able to be verified [53, 55, 212]. The mathematical nature of the techniques avoids speculation about the meaning of phrases which would otherwise be present in a possibly ambiguous natural language description of a system. The formal representations used in the specifications of the system and the properties do not need to be the same, but their underlying logic must be compatible [68]. In essence, formal verification provides a method to either obtain proofs that a system satisfies the purpose it was designed for or to determine the ways in which it fails to do so.

Formal verification is a collection of techniques in which mathematical methods are used to verify systems. It is just one of many verification techniques (see figure 2.1) which also include testing [79, 178] and simulation [18, 153, 234, 242]. Formal verification itself consists of model based verification and other methods. These other methods include theorem proving [11, 116, 118], program derivation [67, 190, 233], static analysis [78, 94, 174, 215, 219], and typed systems [52, 54, 88, 89, 138]. In this thesis we are only using one particular type of model based verification method - namely model checking. We show how model checking fits into the array of software and hardware verification techniques in Figure 2.1.

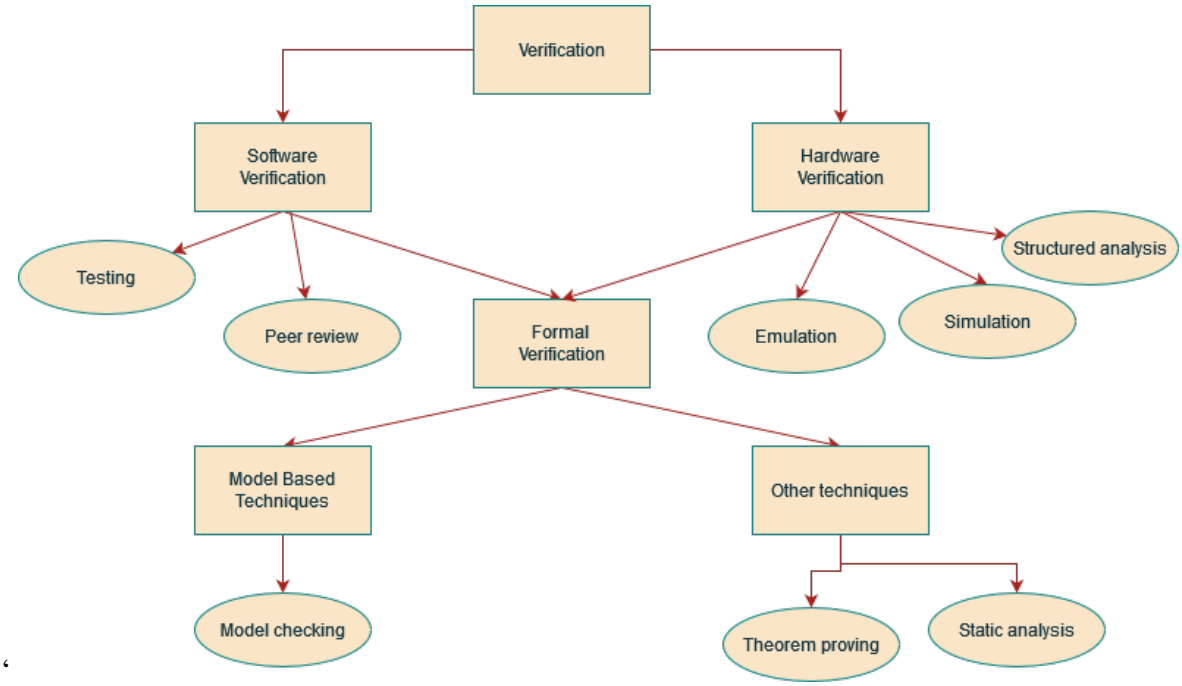


Figure 2.1: Hierarchy of verification techniques and model checking in relation to other verification methods.

Verification is the counterpart to validation: the other process most frequently used to analyse the design of a system [98, 106, 158]. Validation examines the behaviour of a system via simulation or prototyping, whereas verification aims to obtain a mathematical proof for the correctness of the operation of the system.

2.1.1 Model Checking

Model checking is a formal method which allows properties of a system to be checked by building a model of the system and checking whether the model satisfies the properties [37, 41, 77, 204]. It is one of the most successful formal verification techniques and has been widely used in both research and in industry. Model checking is a technique first proposed in the early 1980s by Clarke and Emerson [43] and independently by Queille and Sifakis [196] that is used to verify temporal properties of systems. Unlike theorem proving, the user is not as involved in the verification process. The user only needs to provide formal specifications of the system and property to be checked. The model checking tool then builds the model and, if the model is verifiable, either provides a counterexample showing how the property fails to be satisfied or declares that the property is true. The latter case provides confidence that the property holds for the system by showing that it holds for all executions of the model. In order for this process to be fully automated, the algorithm used by the model checking tool must explore all possible states of the system [46]. For this reason model checking can only be applied to finite-state systems. Many systems of interest are either finite-state or are based on a finite-state control structure [134, 187]. In other cases, where the system is based on an infinite domain, it is often possible to create a finite abstraction of that domain, such that the information necessary for the behaviour of the system is preserved.

In its early days model checking was mostly used to provide verification of hardware systems [68, 140]. However, it has since also been successfully applied to many aspects of software verification [17, 224].

Models are defined using a specification language, and can be checked using a variety of model checkers. Different model checkers have different attributes. For example, they can model the state space *explicitly* or *symbolically* [32, 167], can include quantitative aspects such as time or probability (or neither - in which case only qualitative properties can be considered).

2.1.2 Probabilistic Model Checking

Probabilistic model checking is an extension of traditional model checking that incorporates probabilities and quantitative aspects into the verification process, and is used to verify systems with stochastic qualities. Such qualities are naturally present in many systems, e.g., systems that use randomisation as a part of their operations [157] or systems that involve unpredictable and unknown behaviour, such as the communications delay of a computer network [122].

Probabilistic model checking revolves around the construction and subsequent analysis of a probabilistic model. The probabilistic model used is most typically a Markov chain or Markov process, but can be based on a variety of formalisms. A Markov chain is a mathematical model that describes memoryless stochastic process as a sequence of events, where the probability of each event depends only on the preceding event. Figure 2.2 depicts the relationship between the

most significant formalisms. The formalism used depends on the properties of the system to be

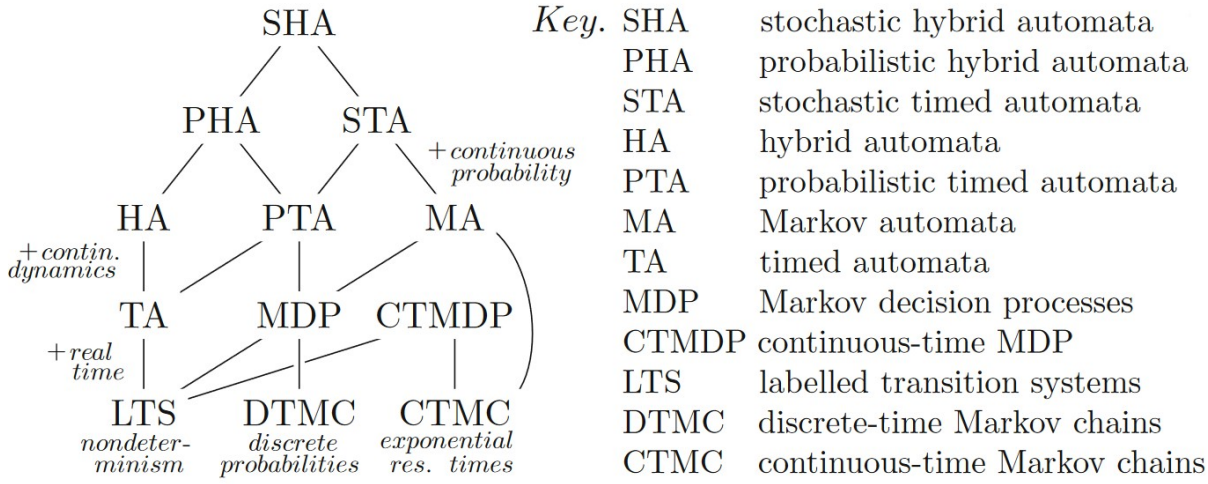


Figure 2.2: Formalisms family tree. All formalisms are extensions of their descendants. Image taken from [103].

modelled. For example, timed automata can be used to represent real-time systems [6]. This in turn influences the choice of model checker as timed automata can only be parsed by certain model checkers. A detailed overview of these formalisms and their expressiveness hierarchy is given by the comparison of probabilistic automata in [217].

Probabilistic model checking requires two inputs: a description of the system to be analysed, typically given in some high-level modelling language; and a formal specification of quantitative properties of the system that are to be analysed, usually expressed in variants of temporal logic. Transitions are labelled with quantitative information regarding the probability and/or timing of the transition's occurrence. Markov chains can also be augmented with rewards, used to specify additional quantitative measures of interest [149]. Once this model has been constructed, it can be used to analyse a wide range of quantitative properties of the original system, relating for example to its performance or reliability. In contrast to, say, discrete-event simulation techniques, which generate approximate results by averaging results from a large number of random samples, probabilistic model checking applies numerical computation to yield exact results. Probabilistic model checking often relies on symbolic representation of states, typically through the use of Binary Decision Diagrams (BDDs) (see Chapter 3 for a definition). Using a representation of a group of states rather than systematically exploring all states results in a more efficient storage and computation. Symbolic methods have been effective in the design of control systems [101, 211]. We have chosen to use a symbolic approach in this thesis because of its efficiency and because probabilistic reasoning is most appropriate for our main case study – a protocol for wireless sensing and control.

Statistical model checking [155, 156] is a simulation-based approach to verifying quantitative properties which allows probability thresholds to be obtained for properties. The key idea is

to observe some of the system executions via a monitoring procedure [19], and to use hypothesis testing to decide whether the executions provide enough statistical significance for the confirmation or disproval of a property [245]. Statistical model checking differs from the numerical approaches described above in that it does not provide a guarantee for a correct result due to its simulation-based nature; however, it is possible to provide a bound on the possibility of making an error. It has been used in CPSs as it is helpful when it is inconvenient or impossible to obtain a concise representation of a global transition [47]. The main limitation for statistical model checking are properties concerned with rare events, i.e. whose satisfaction probability is very small.

2.1.3 Model Checkers

Model checkers are automated tools used to perform formal verification of systems. They systematically explore all possible states or behaviours of a system model and check if it satisfies a given set of properties or specifications [186]. The basic methodology behind a model checker is shown in Figure 2.3: the system that is being analysed is used to construct a state-transition formal model M, the desired properties of the system are described using a logic formula ϕ, and the model checker determines whether the formula ϕ is modelled by the model M, $M \models \phi$. If $M \not\models \phi$, then the model checker produces a counterexample that presents an instance where the formula is violated. Model checkers are widely used in various domains, including hardware and software systems, protocols, concurrent systems, and distributed systems.

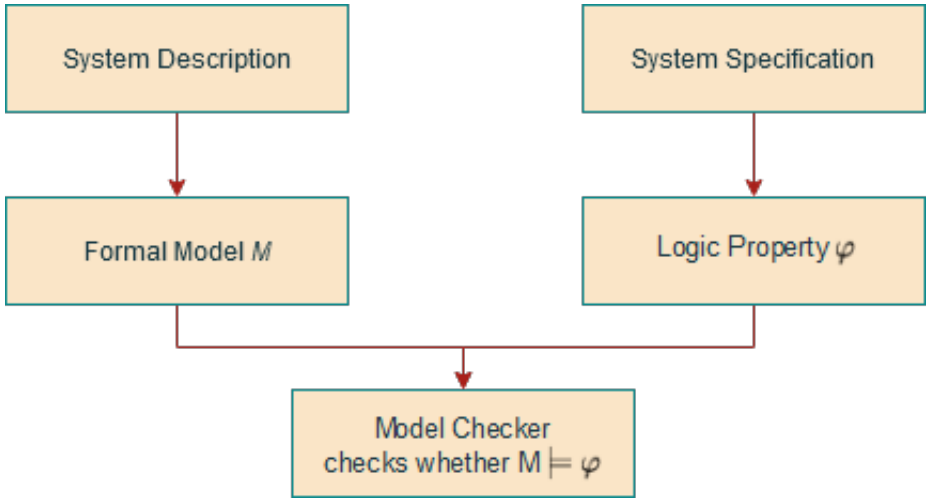


Figure 2.3: Basic model checker methodology

Model checkers can be broadly classified into two categories: qualitative and quantitative. Qualitative model checkers support properties that refer to states never being reached or to the order in which events occur [95]. These properties contain no quantitative aspects such as probabilities or timing constraints. On the other hand, quantitative model checkers are capable of analysing quantitative aspects of a system, such as probabilistic or time-dependant behaviour

[151]. They are capable of determining numerical values or probabilities associated with specific states being reached, for example the total *reward* accumulated during execution of a model. There is a wide variety of model checkers available, each with its own strengths, supported formalisms, and verification techniques. Some examples of model checkers are listed below, and some of their characteristics are compared in 2.1:

- SPIN [114] is a well-known non-probabilistic model checker for concurrent systems. It supports the verification of properties expressed in Linear Temporal Logic (LTL) and uses a depth-first search algorithm to explore the state space. SPIN is widely used in the analysis of concurrent and distributed software systems [115, 126].

- The Alloy Analyzer [121] is a model checker that focuses on software analysis and design. It uses a declarative modelling language based on first-order logic constraints. The Alloy Analyzer employs a SAT-based solver to explore the state space of the system model and check properties. It is often used for analysing software architectures and system behaviour [13, 137, 142, 224].

- NuSMV [39] is a symbolic model checker that is a reimplementation and extension of the SMV model checker. NuSMV models are closely tied to their expression as boolean formulas, and allow for hierarchical specifications which make use of reuseable components. The model checker supports both asynchronous and synchronous communication. NuSMV is often used in the formal analysis of hardware systems and protocol specifications [50, 168].

- UPPAAL [20] is a model checker specifically designed for the verification of real-time systems. It supports the modelling and verification of systems with both discrete and continuous behaviour, such as timed automata. Typical application areas are those where timing aspects are critical, such as real-time controllers and communication protocols in particular [17, 123, 169].

- PRISM [143] is a powerful symbolic probabilistic model checker that specialises in the analysis of probabilistic systems. It supports various formalisms, including Markov Chains, Markov Decision Processes, and Probabilistic Timed Automata. PRISM provides a variety of analysis techniques, whose implementations are partly explicit (based on sparse matrices) and partly symbolic (based on Binary Decision Diagrams). It supports advanced techniques, such as multi-objective model checking [195] and parameter synthesis [194]. The tool is highly interconnected by language translators and the Hanoi Omega-Automata format [12]. PRISM is probably the most widely used verification tool on our list and have been used in a variety of areas, including analysis of satellite positioning systems [163, 164], the rates of biological processes [152], and the behaviour of autonomous agents [113]. PRISM will be introduced in detail in Section 3.5.1.

Language	Supported Formalisms	Module communication	Engines
SPIN	Discrete (finite-state systems)	Asynchronous, message passing, rendezvous	Explicit-state, partial order reduction, bit-state hashing
Alloy	Discrete (relational models)	Both (declarative relational modelling)	SAT
NuSMV	Discrete (finite-state systems)	Primarily synchronous	BDDs, SAT
UPPAAL	Timed (timed automata)	Asynchronous with rendezvous	Difference-bound matrices, polyhedral reasoning
PRISM	DTMC, CTMC, MDP, and PTA	Via synchronisation	MTBDD, Sparse, Hybrid, Explicit
STORM	Discrete- and Continuous-Time Markov models DTMC, CTMC, MDP, and MA	Via synchronisation	Automatic engine selection Sparse, Hybrid, Exploration DD, Abstraction-Refinement
ePMC	DTMC, CTMC, MDP, and stochastic games	Primarily synchronous	Layered iteration approach Büchi and Rabin automata

Table 2.1: A selection of model checkers and their corresponding modelling languages, and their characteristics. Based on information from [186].

- STORM [112] is a probabilistic model checker that can handle both discrete and continuous-time Markov Chains as well as Markov Decision Processes and Markov automata. Similarly to PRISM, it supports advanced techniques, such as multi-objective model checking [195] and parameter synthesis [194], as well as both symbolic and explicit state space representations and a mixture of the two. The direct usage of a wide array of modelling languages is supported as an input, including the PRISM language. STORM is commonly used for analysing large-scale, complex probabilistic models [132, 209].

- ePMC (An Extendible Probabilistic Model Checker, previously known as IscasMC) [81] supports Markov Decision Processes, Markov Chains and stochastic games, and allows specification of properties in PCTL*. ePMC excels in efficient computation of linear time properties [104] and boasts high modularity and extensibility of the tool with plugins for new features. In this way, it can be extended for specialised purposes allowing the verification of niche systems [75, 82]. It also supports direct usage of the PRISM language as an input.

Choosing an appropriate model checker depends on various factors, including the characteristics of the system being analysed and the properties to be verified. This can be a challenging task without in-depth knowledge of all available tools. The QComp 2019 competition [103] was held as a friendly forum to compare tools analysing quantitative formal models. Its results showed that each tool had different capabilities and different tools excelled at different bench-

mark instances, with no one tool taking the lead in terms of performance. The PRISM model checker was regarded, however, as the foremost in terms of usability. It has extensive online documentation, provides a graphical user interface, and ease of installation with only Java as a dependency. These, coupled with our previous experience with PRISM, are the reason that this thesis mostly focuses on PRISM as the model checker of choice. The second iteration of the competition, QComp 2020 [30], saw a significant increase in the modelling capabilities of a number of the participating tools. This time the STORM model checker dominated the competition; however, the authors noted that its new automatic engine selection causes it to not fit well with the competition design. Overall, model checkers are an active area of research with new tool capabilities being steadily introduced. We discuss PRISM in detail in Section 3.5.1.

2.1.4 State space explosion

The state space explosion problem is a challenge encountered in model checking: the state space of a system can grow exponentially with the number of components, variables, or events. For example, suppose that a system is composed of m processes, each made up of n states. The composition of those processes can have up to n^m states. Likewise, a system consisting of an n-bit counter has 2^n states. Model checking involves exhaustively exploring all possible states of a system model to verify if certain properties hold, so this exponential growth can make the task computationally infeasible. This problem is referred to as the *state space explosion* (or *state explosion*) problem in the context of model checking and affects all model checkers. Complexity theory has been used to show that, in the worst case, this problem is inevitable [45]; however, there has been a number of techniques developed that address this issue. Some techniques aim to reduce the state space of a system before the verification process begins by analysing the system's structure or properties. Others focus on pruning portions of the state space that are known to be irrelevant to the property in question during the verification process. We review a few of these techniques below.

Symbolic model checking techniques operate on symbolic representations of the system model instead of explicitly enumerating all possible states. A set of states is represented by a BDD which results in a representation which is often exponentially smaller. BDDs were first introduced by Lee [154] and then popularised by Akers [2] as a compact and efficient way of representing Boolean functions. They are later extended via the addition of an ordering of their variables [29]. As the size of the BDDs is greatly impacted by the ordering used - ranging from linear to exponential - the way of selecting an ordering has been a significant research interest. Determining the best variable ordering prior to constructing the BDD is an NP-hard problem [141]. A number of ordering heuristics have been created [108] to obtain an initial ordering and reordering algorithms [227] that obtain a BDD representation with a minimum number of nodes for a specific subclass of models. Symbolic model checking has been used to model systems much larger than those possible with explicit model checking [31, 32].

Partial order reduction techniques [92] eliminate redundant interleavings of events in systems with an asynchronous composition of processes where the order in which certain events occur does not affect the satisfiability of the property being checked. This technique is based on the observation that the exploration of some paths can be avoided without changing the resulting verification outcome. Such paths exist because events that are independent of each other can be executed in any order without impacting the result. Figure 2.4 shows two independent paths $(s \rightarrow s_a \rightarrow s')$ and $(s \rightarrow s_b \rightarrow s')$ where transitions update the value of a local variable `var`. In this

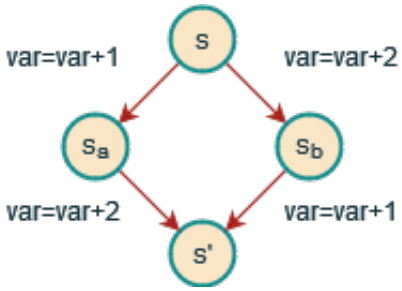


Figure 2.4: Independent paths that reach the same end state.

case it would be sufficient to only explore one of these paths during the model checking process. One of the biggest challenges for partial order reduction is that the reduction must be performed on the fly [186]. Which transitions can be ignored must be decided during the construction of the transition system as it would be counter-productive to first construct the whole transition system and then prune it. Therefore, decisions need to be done locally, without global knowledge of the transition system. Selecting a subset of available paths allows us to construct and perform verification on a smaller state space which uses the selected paths as representatives of the equivalence classes of all paths [129]. If a counterexample for a property exists in the full state space, it must also be present in the reduced state space. Traditionally, partial order methods have been used by the enumerative algorithms of explicit state model checkers, and typically for asynchronous systems; however, they have also been applied to symbolic model checking [5]. There exist several methods that perform the selection of paths: ample sets [185], stubborn sets [226], and persistent sets [93]. While these methods were developed independently, they are similar in some respects, and have been used together [99]. Some partial order reduction variants also take into account the state from which transitions are performed to refine the reduction process [130]. Partial order reduction has been recently used in schedule-abstraction graphs, a reachability-based response-time analysis technique [198]. It was shown that partial order reduction led to an impressive 98% reduction in the number of states explored, causing a runtime reduction of five orders of magnitude.

Symmetry reduction techniques aim to eliminate redundant states that arise from symmetries in the system. They exploit structural symmetries to collapse or merge equivalent states. We will explore these in more detail in Section 2.1.5.

Knowledge of properties of the state space has not only been used in relation to symme-

try. A class of systems, called *degenerative*, that have the property that a model of the system will eventually behave like a model of a smaller system, is investigated in [170]. In this paper the behaviour of the Firewire leader election protocol on a given number of nodes is analysed. Abstraction and induction techniques are used to show that properties that hold for a small number of nodes will also hold for a larger number of nodes. Another induction property is shown on a class of quantitative LTL properties for degenerative systems [95]. In [36] abstraction is used to prove general results about a telecommunications protocol involving any number of processes. The abstraction technique involves creating an abstract process which represents a sum of any number of processes. This way any system can be described as having a fixed number of concurrent processes and one abstract process that holds information about all of the other processes.

Bounded model checking is a technique that relies on a SAT solver to perform an exhaustive search for counterexamples of limited length. It was introduced as an alternative to symbolic model checking without BDDs [23]. It has been shown to perform better than BDD based verification techniques in some cases and to require less user configuration (as opposed to the ordering of BDD variables) [42]). Drawbacks of this approach include the fact the types of properties that can be verified are limited, and as counterexamples are of bounded length, the verification approach is incomplete. Therefore, bounded model checking is better suited to finding bugs in a system, rather than proving correctness, and as a result has found successful application in hardware verification [24]. An overview of recent advancements in bounded model checking are highlighted in [229].

Other state space reduction techniques include counterexample guided abstraction refinement (CEGAR) [44] and program slicing [222], but we do not use them in this thesis.

The techniques to tackle state space explosion mentioned above have also been applied to a probabilistic context. Partial order reduction for probabilistic model checking has been introduced simultaneously in [16] and [49]. Those works were focused on model checking LTL properties on Markov Decision Processs (MDPs). The probabilistic variant of this technique is later revised in [15] by ensuring that distributed schedulers are preserved. The most prominent application of symbolic model checking to probabilistic systems is PRISM. Dynamic symmetry reduction has been applied to PRISM as a way to circumvent the problem caused by generating all orbits of the symmetry relation in probabilistic systems [232]. Abstraction refinement techniques have been applied to probabilistic software [128] to target quantitative properties of software. Experimental results have shown that the approach performs very well in practice, being successfully used to verify networking software with complexity beyond the scope of other probabilistic verification tools.

These techniques can be used individually or in combination to mitigate the state space explosion problem in model checking. The choice of technique depends on the characteristics of the system model, the properties to be verified, and the available computational resources.

Each technique has its trade-offs in terms of precision, scalability, and the types of systems they can effectively handle.

2.1.5 Symmetry reduction

A number of different approaches exist to exploit symmetry when modelling a system consisting of multiple identical components. A common idea is to abstract the model in such a way as to allow for computations for the identical components to be reused rather than redone.

Symmetry reduction [40] is a common strategy to reduce the state space of models consisting of identical components. The inherent symmetry of the original system will be reflected in the state space of the constructed model. Model checking search techniques involve the exploration of the state space of the model. Therefore, symmetrically equivalent areas of the state space can be searched only once by leveraging knowledge of the symmetry of the system. There are a number of approaches and techniques that have been proposed and used for symmetry reduction [172]: the construction of a reduced state space where symmetrically equivalent areas of the state space have been identified as one. The exploitation of symmetry has also been investigated in the context of process networks [34].

The concept of *generic representatives* is introduced [71] as a way to achieve symmetry reduction. This method consists of translating the source code for a model into that for a reduced model, which can still be explored using model checking algorithms.

A related technique is known as a *population* approach whereby the effect of actions on populations of components/agents are considered, rather than on individuals. A model representing the interactions of a finite population of identical finite-state agents is presented in [8]. This model consists of *population protocols* and *population configurations*. An application of population models for a distributed robot protocol is examined in [85]. The resulting models are used to complement simulation results in order to facilitate the logistics of swarm deployment. This concept is extended in [86] for non-centralised protocols by providing modelling support for agent synchronisation. A holistic approach to the multiple layers of population abstraction is given in [87].

2.1.6 State space reduction tools

General purpose state space reduction tools have not been well established in the field of model checking. Typically state space reduction techniques such as those described in Section 2.1.4 are used to create a state space reduction algorithm which is then implemented for a specific model checking tool/process. As a result, model checkers typically have state space reduction techniques built in that are distributed packaged together with the model checker and are triggered either automatically or through the use of optional command arguments. For example, *program slicing* [222] is a well-established state space reduction technique, whose basic idea is to elimi-

nate details of the program being verified that are irrelevant with respect to the property that is being analysed. This technique has been labelled by the more general name of "state space reduction" in the literature [76] but in this thesis we will use the term *program slicing* to distinguish between this and other techniques that result in a smaller state space. The AgentSpeak(L) BDI logic programming language [199] is used for the programming of multi-agent systems; e.g., autonomous rovers for planet exploration. Model checking techniques and tools for AgentSpeak [26] are based on translating AgentSpeak specifications into either Java or Promela (for use with the JPF model checker or the SPIN model checker, respectfully). Property-based slicing has been used in the creation of algorithms specifically targeted at AgentSpeak programs [25].

PRISM-symm [144] is based on an efficient algorithm for the construction of quotient models. It uses a symbolic implementation based on the multi-terminal binary decision diagrams data structure [14, 83]. As the name implies, this tool is closely related to the PRISM model checker and performs a model-level reduction built into the symbolic implementation of PRISM. Its approach is based on dynamic symmetry reduction approaches for non-probabilistic models [70, 231].

Generic representatives are used as a way to apply counter abstraction [65] in the prototype tool GRIP (Generic representatives in PRISM) as a way to convert symmetric PRISM models into symmetry reduced form. Limitations of GRIP in regards to complex models are addressed by introducing a richer language, which aids the application of generic representatives [59]. The benefit of this approach is that it works at language-level, so while GRIP was implemented with PRISM as its main target, it is possible to be used with any other model checker which accepts the PRISM modelling language as an input. GRIP will be discussed in detail in Chapter 6.

There are two tools that provide symmetry reduction for the SPIN model checker: SymmSPIN [27] and TopSPIN [63, 64]. Both of these tools require the user to additionally specify the symmetry present in the model, and will provide unreliable output should the declared symmetry be mistaken. There has been work done to automate the symmetry detection process [60], and SymmExtractor, a tool for the automatic detection of symmetries present in Promela specifications has been created [61] and evaluated [62]. Both tools have been shown to be unable to exploit the symmetries of all SPIN models [27, 158].

The UPPAAL model checker has had a prototype symmetry reduction tool constructed [111]. This tool was shown to obtain promising results by allowing the expression of total symmetries using a scalarset approach (which were originally developed for the Murphi language [100]) with the intent to be extended to apply to other symmetry types (such as the ring symmetry present in token ring protocols). It was integrated into UPPAAL 4.0 [20]. The recently released UPPAAL 5.0 does not include any updates to the symmetry reduction tool. UPPAAL's symmetry reduction has been successfully applied to the Zeroconf protocol [90] and a leader election algorithm [188].

2.2 Wireless Sensor Networks

WSNs are a class of networks composed of numerous small low-cost wirelessly interconnected sensor nodes [243]. These sensor nodes are typically power-constrained and equipped with sensing and processing capabilities, which allows them to gather and process data from their environment. The data is then transmitted to a gateway node or another node within the network (which in turn transmits the data towards the gateway node). The collected data is used to observe and to respond to events that occur in the environment that is being sensed.

The sensor nodes of a WSN are able to cooperatively sense, control, and transmit data about the environment that is being sensed. They typically are small low-cost devices capable of being deployed in a wide range of environments: e.g., buildings, fields, cities, forests, etc. Depending on the use case, sensor nodes can gather information about a variety of physical and environmental quantities: e.g., temperature, motion, light, atmospheric pressure, air quality, humidity, sound, location, etc.

As the name suggests, communication in a WSN is wireless, which allows nodes to establish ad hoc connections. This results in a self-configuring network that does not rely on a pre-defined structure. Furthermore, WSNs offer a reduction in deployment costs compared to their wired counterparts. Data transfer is a collaborative process: sensor nodes often relay each other's data to reach the intended destination. Multi-hop communication, the process in which data is transmitted from one node to another until it reaches the desired destination, is frequently employed. The topology of a WSN is illustrated in Figure 2.5. Sensor nodes gather data about the system and transfer that data to a gateway node. The gateway node then propagates the data through the Internet to the user. The self-configuring nature of the network has presented a problem in designing the communication between sensor nodes. Different routing protocols for WSNs are presented in [3, 131].

During the last three decades, WSNs have held increasing interest in both research and industrial spheres, alongside a continual increase in use domains and application scenarios [241]. Applications of WSNs include agriculture, healthcare, wildlife tracking, military, disaster management, industrial automation, home automation, traffic monitoring, etc. For example, sensor networks were found to be the ideal platform for the surveillance of endangered bird species [28]. WSNs play a fundamental role in the enabling, establishment and spread of Internet of Things (IoT) technologies [240].

WSNs often face challenges such as limited energy supply, limited memory and processing power, unreliable and limited communication, environment coverage and data delivery time constraints. Hardware limitations prompt the need for data sensed by the devices to be transmitted to the gateway node with the least resource utilisation. Spatial and temporal constraints depend on the nature of the sensed physical phenomena: it is likely that the delivery of data would be restricted by time as data might quickly be rendered invalid in dynamic systems. As a result, various techniques and protocols have been developed to improve data reliability, reduce energy

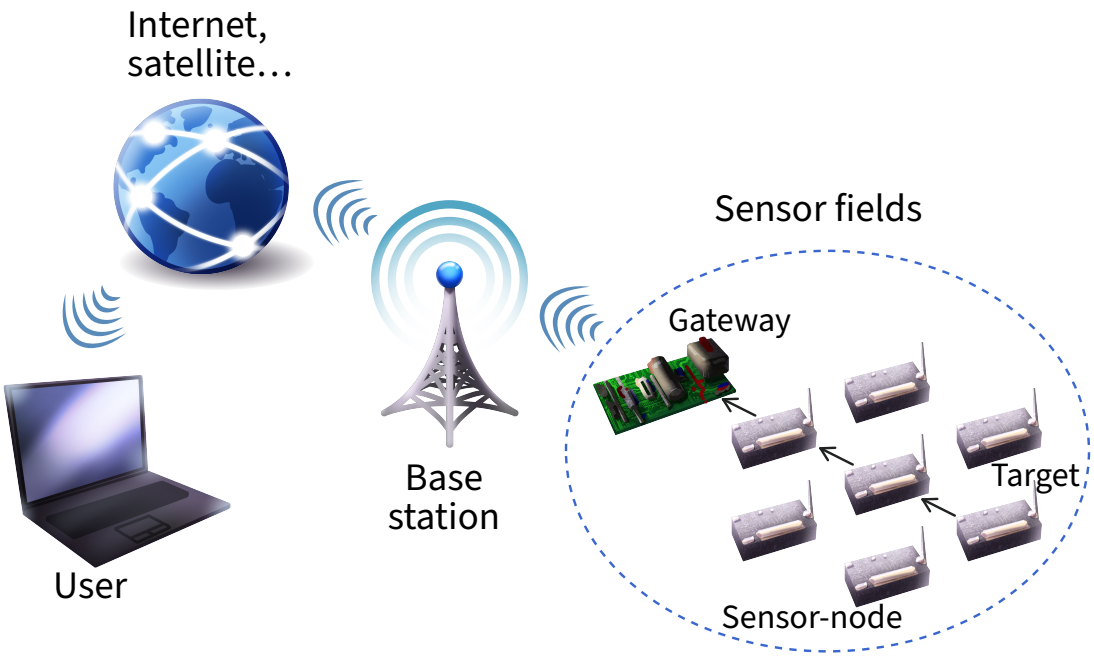


Figure 2.5: A WSN topology.

consumption and optimise intra-network communication. Network deployment is a research topic that gathers large amounts of interest as the number and locations of sensors established during deployment determine the topology of the network [210]. Many properties of interest directly depend on this network topology, so the performance of a WSN is highly influenced by its deployment. For the rest of this thesis, however, we focus on challenges faced by WSNs after their deployment. Many surveys of WSN technologies and the challenges faced by them are available [4, 192, 201].

2.2.1 Characteristics of Wireless Sensor Networks

WSNs vary depending on their sensors' characteristics and on their application requirements. Based on the specifics of a WSN, sensors cooperate according to various models and architectures. We go over some of the most frequent ways of classifying a WSN:

Depending on its size, a WSN can be classified as a small, medium, large or very-large-scale network. The size depends on factors such as the characteristics of the sensors, the goals of the sensor network and its region of interest. In practice, the number of sensor nodes in a WSN ranges between tens to thousands of sensors.

The type of a WSN is also determined by the sensors it consists of. A *homogeneous* WSN contains sensors that are considered identical with respect to the purposes of the network (i.e. they are equally capable in terms of sensing, communication, processing power, memory storage, and energy supply). *Heterogeneous* WSNs allow the presence of nodes with elevated or di-

minished capabilities, typically ones that perform a specialised or an additional function. These sensors may not only differ in their sensing and data processing roles, but also in their communication aspects. Such heterogeneous WSNs are called *hierarchical* (as opposed to *flat*) and often distinguish their nodes into classes according to their communication capabilities. For example, a sensor might be designated as a cluster-head and be responsible for managing communication between sensors in its cluster and other clusters. In this thesis, we will focus on homogeneous sensor networks.

Depending on the location of sensors, a WSN may be static, mobile, or hybrid. Static WSNs consist of stationary sensor devices that do not move after they are deployed. As mobile devices have become more popular as a result of technological advances, so have mobile sensor devices. A *mobile* WSN consists of sensors which are able to move on their own after deployment, while a *hybrid* WSN consists of both stationary and mobile sensors. Mobile WSNs offer greater versatility as they can respond to changes in the environment [1]. The addition of some mobile sensor nodes can improve the flexibility of a WSN by allowing those nodes to respond to changes in network topology.

Sensor nodes in a WSN transmit their data to a gateway device (often called a sink), but networks differ in the way that is achieved. Single-hop WSNs are more topologically restrictive and require nodes to transmit their data directly to the sink, while multi-hop WSNs allow messages to be relayed through multiple nodes to the sink. A multi-hop WSN can be either flat or hierarchical.

These WSN characteristics, combined with restrictions of the sensor devices (e.g. limited memory, processing power, communication capabilities), result in different design and performance criteria for the communications protocols used. These communication protocols need to manage and control all aspects of communication: from the topmost application and transport layers responsible for packet construction and end-to-end transmission of packets, to lower layers for routing of data, and to the lowest layer of accessing the physical medium [124]. Figure 2.6 depicts an overview of the network architecture in WSNs.

Communication protocols for WSNs can be classified in one of three types depending on what aspect of the communication they are concerned with: transport layer protocols, network layer routing protocols and data link layer Medium Access Control protocols [124].

The end-to-end transmission of data packets is handled by transport layer protocols. These protocols are classified by whether they provide guarantees for reliable data delivery through retransmission of packets or if they support unreliable data delivery [200]. Furthermore, they carry out congestion control, in either a centralised or distributed manner, by determining the appropriate transmission rates of sensor devices. [124] provides a comparison of these aspects for a variety of transport layer protocols for WSNs.

Routing protocols in the network layer are an area that has seen large research interest. They determine how messages propagate through the network in a way that ensures efficient perfor-

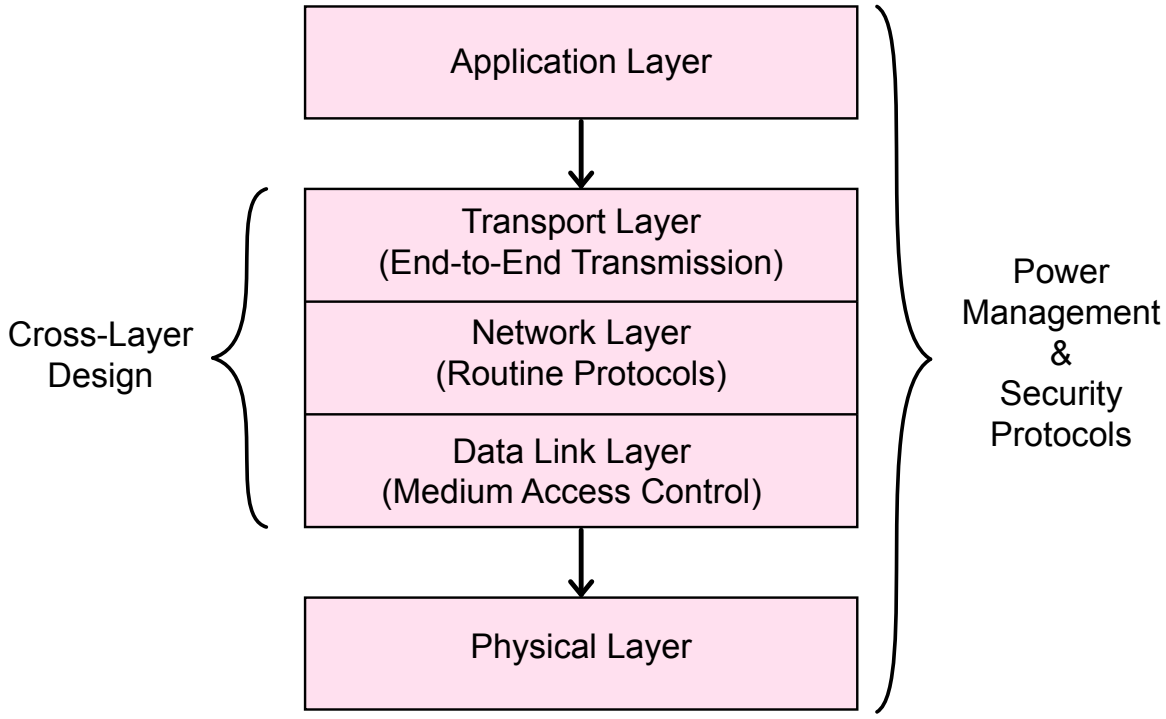


Figure 2.6: Communication network architecture of a Wireless Sensor Network. Image is based on image from [124].

mance as well as meets the necessary bandwidth, fault-tolerance, delay and energy constraints. Each routing protocol is designed according to the characteristics of the WSN that it will be used by: the routing strategy (reactive, proactive, hybrid, etc.) [125]; the topology of the network (flat or hierarchical) [230]; the communication entities (data-centric or node-centric); the data sampling and retrieving approach used (e.g. query based routing; QoS-based routing) [117]; and a large variety of other operation-based attributes [125, 235]. Security for routing protocols is an active area of research [214, 246] because typical use of symmetric and asymmetric key cryptography are often too resource-intensive for resource-limited WSNs [248]. An in-depth comparison and analysis of routing protocols for each different type of WSNs is presented in [131].

The lowest level medium access control (MAC) protocols establish how devices interact with the shared physical (wireless) medium and the strategies that are employed to avoid collisions. Typical strategies involve decomposing the shared medium (radio channel) into multiple disjoint channels or slots where devices can transmit with a guarantee that collisions will not occur. Examples include Time Division Multiple Access (TDMA) where slots are separated temporally [205], Frequency Division Multiple Access (FDMA) which divides the shared medium in frequency bands [131] or Code Division Multiple Access (CDMA) which employs a more complex approach utilising unique code assignment to devices [120]. Traditionally, only a single simple channel is used to employ these strategies in order to conserve energy [105]; however, recent

research has shown that the use of multiple radio channels can also be useful.

MAC protocols are broadly organised into two classes: schedule based or contention based protocols. Schedule based ones assign a portion (of time, of frequencies, etc.) of the shared medium to each node, so that at each portion there is only one transmitting device making it impossible for collisions to occur. On the other hand, collision based MAC protocols do not produce such a schedule but instead assume that some collisions will occur and define a back-off procedure that devices must follow to resolve those collisions. Another important aspect of MAC protocols is the concept of duty cycling, which is an energy conservation process employed by sensor devices [221]. This is achieved by devices periodically going to a sleep mode and turning off their radios. In this thesis we focus on formal verification of MAC communication protocols, so we present some MAC protocols in greater detail below (Sections 2.2.2, and 2.2.3).

2.2.2 Ctrl-MAC

Ctrl-MAC [22] is a TDMA protocol that was developed in the design of a slow-loop controller and communication system for large area infrastructure. Examples of this sort of infrastructure includes smart urban water distribution systems, precision agriculture and the electrical power grid. We refer to these systems as CPSs.

Cyber-Physical Systems are the combination of a communication system (such as a sensor network) that gathers data about the environment and a control system that performs actions to interact with the environment in an appropriate way based on the gathered data. WA-CPSs are a subtype of CPSs which are designed towards the sensing of a geographically large environment. The communication system used in WA-CPSs is typically a WSN as the wireless connection is a low-cost option for communication over the required distances.

Ctrl-MAC was created with the goal to allow sensor systems to outscale the previous state-of-the-art LPWA MAC protocol LoRaWAN [48]. It employs a novel co-design approach called *Control Communication Co-design (C^3)* which considers communication-and-control systems at two stages: design-time and run-time. At design-time the effect control parameters and communication parameters have on each other is considered. The parameters are then chosen such that the resulting system is globally exponentially stable during run-time, without any subsequent updates to the parameters. A co-design approach has been shown to be vital in the construction of a WA-CPS because of the influences the communication system and control system have on each other [181]. The resulting communication protocol achieves this result by placing control of communication access at the gateway node: i.e. the gateway schedules data transmissions to only those nodes that request it. The C^3 approach has provided the necessary reduction of delays and the improvement of reliability [22]. Ctrl-MAC is the main example used in this thesis, and we discuss it in detail in Chapter 5.

2.2.3 2C protocol

In this thesis we are concerned with applying model checking to analyse the Ctrl-MAC protocol. A comparable WSN protocol which has been formally verified is the 2C-WSN protocol, which is used in the contention phase of IEEE 802.15.4 [165]. 2C-WSN has only one request slot, sensors choose at random whether they are backed off or not after a collision, and all back-off periods last until a cycle without a collision occurs. The model given in [165] scales to 40 sensor devices. We view 2C-WSN as a simplified version of Ctrl-MAC as it uses a similar transmission request based approach but without a complicated back-off assignment. As we will draw connections between Ctrl-MAC and 2C-WSN (see Chapter 5), we introduce it in some detail here.

The 2C protocol is a simple full feedback sensing window random access algorithm designed to solve the Channel Multiple-Access problem for strict delay messages [183]. Binary collision-noncollision feedback per slot together with full knowledge of the feedback history is used to provide an upper bound on the transmission delays on messages. It has been shown to allow for the same throughput as Capetanakis' dynamic algorithm [38], while being more resistant to feedback channel errors and inducing lower transmission delays for arrival rates above 0.30. Contrary to Capetanakis' protocol placing unsuccessful users on an infinite cell stack (similar to Ctrl-MAC's infinite cell queue), the 2C protocol only uses a 2-cell stack by grouping all of the unsuccessful users together [184]. A limited sensing version of 2C is presented in [182], where devices observe the feedback of the channel (i.e. listen for success/failure messages) from the time they generate a packet to the time that packet is successfully transmitted. Limited sensing is possible if a user can detect in finite time whether a collision resolution is in progress [91]; however, that implicitly adds an overhead to each packet transmission as channel feedback needs to be sensed until the completion of a collision resolution interval. Despite this overhead, limited sensing algorithms have been shown to obtain the same throughput as their full sensing counterparts [119], as it was found that only the transmission delays were impacted. All of the protocols mentioned above rely on a key transmission property: no new packages are allowed to contend for a transmission slot before the current collision resolution is completed. As this is a property that largely impacts the average delay of transmissions, Ctrl-MAC aims to enable limited sensing without defining global collision resolution intervals. Limited sensing is still required as users need to transmit at the appropriate time slot; however, the sensing period is up to the first Request-Reply message as that would allow sensors to synchronise with the slotted time. As a result the length of time users with newly generated packets are required to sense the channel feedback is less than or equal to the duration of one Request-Reply cycle whereas a global collision resolution would require multiple such cycles.

The 2C protocol has been adapted to a wireless communication medium by the 2C-WSN (two cell wireless sensor network) protocol (also sometimes referred to as two cell sorted wireless sensor network, 2CS-WSN) [205]. Contrary to wired networks where it is easier to detect collisions, in wireless networks it is necessary to infer that a collision has happened and is usu-

ally done by assuming that a collision has happened when a reply to a request does not arrive. 2C-WSN makes use of the Parent Available message to establish a tree structure between all colliding sensors. It has been designed to be used by the contention phase of IEEE 802.15.4.

While the developers of 2C-WSN evaluated their proposed protocol using simulation in the wireless sensor networks simulator Castalia [177, 205], formal analysis of 2C-WSN has since been done. A Markov Decision Process has been used to provide proofs for properties for individual sensors [206]. Some different variants of 2C-WSN have been proposed as improvements as a result of other protocol formalisations [207].

2.3 Verification of Wireless Sensor Networks

As sensor technology continues to develop, WSNs find increasingly widespread application; however, each new use contains a new set of challenges. Analysis of sensor networks is crucial for the discovery of system errors, especially for design flaws that must be addressed before sensor network deployment. Additionally, if errors occur, it is desired that we ensure that the sensor systems fail in a manner that is non-destructive towards the system that is being monitored and controlled. Specification and verification of sensor networks are highly non-trivial tasks due to the inherent properties of sensor networks [66]. In particular, WSNs are highly affected by their environment, especially those deployed on critical infrastructure [236]. The performance of sensor nodes is influenced by external factors such as changes in temperature and precipitation, and potential sensor node failure can worsen the behaviour of the entire WSN. To diminish the effects of these potential problems, it is necessary that the software controlling the WSN is reliable. Dynamic environmental factors are one of the most significant reasons for the failure of WSNs and probabilistic models are critical for the quantitative analysis needed to prevent such failures [151]. For WSNs it is necessary to model not only the sensor devices and the relationships between them, but also how those relationships evolve over time due to environmental factors. Consequently, formal methods have been frequently used for the analysis of WSNs and, more generally, IoT.

Most of the model checkers listed in Section 2.1.3 have been used to perform some form of formal verification of WSNs. The Alloy Analyzer has been used for the security analysis of an authorisation toolkit for the Internet of Things [133]. SPIN has been used to aid in the development of Insense, a tool used to verify the correctness of WSN applications [213]. The UPPAAL model checker has been used to carry out formal verification of real-time WSN protocols [176]. PRISM has been used to analyse security risks of IoT systems [175], as well as slot allocation and contention resolution protocols for WSNs [80]. PRISM has been used to analyse survivability properties of WSNs [189]. It has also been used to verify a wireless sensor network architecture [173] as well as to verify an array of protocols for WSNs [237]. Additionally, PRISM has been used in the construction of a run-time verification framework for sensor

networks [239].

There are also numerous approaches and formalisms that have been introduced with the purpose of aiding sensor network verification. The use of Graph Transformation Systems and Bigraphical Reactive Systems is explored in [9]. Process algebras have been created with the goal of describing the forms of interference between processes of wireless systems [73, 150]. In wireless communication such interference occurs when two transmissions simultaneously reach the same location. Coloured Petri Nets have been successfully used to create models that express the power consumption caused by communication protocols used by WSNs [51] in order to enable the evaluation of a WSN lifetime. Probabilistic reactive modules have been used to verify the synchronisation of nodes in a WSN with respect to the environmental effects on a sensor's hardware clock [236]. The concepts of *frames of reference* and *frames of function* are introduced in [35] as a way to organise models and ensure that all key features of the system are being captured.

Chapter 3

Preliminaries 1: Model checking

In this chapter we introduce the concepts, formalisms, tools and techniques used throughout the rest of this thesis. First, we define Discrete Time Markov Chains, which are used to model discrete systems.

3.1 Discrete Time Markov Chains

Discrete Time Markov Chains (DTMCs) are memory-less stochastic processes, i.e. infinite sequences of random variables $X_0, X_1, X_2, \ldots, X_t, \ldots$, in which the probability of the state of the next random variable only depends on the current state. This memory-less property is called the Markov Property, and it can formally be written as

$$P(X_{t+1} \mid X_t, X_{t-1}, \ldots, X_0) = P(X_{t+1} \mid X_t)$$

The domain of the random variables is some state space $\mathscr{X}$, i.e. $X_t \in \mathscr{X}$ for all t. We note that the probabilities in the statement of the Markov property are normalised because the transition from a state X_t to any possible state X_{t+1} is 1, i.e. $\sum_{X_{t+1} \in \mathscr{X}} P(X_{t+1} \mid X_t) = 1$.

Definition 1. *A Discrete Time Markov Chain D is a tuple $(S, s_{init}, \boldsymbol{P}, L)$ where: S is a set of states called the state space, $s_{init} \in S$ is the initial state (often labelled s_0), $\boldsymbol{P} : S \times S \to [0,1]$ is a transition probability matrix, and $L : S \to 2^{AP}$ is a function labelling states with Atomic Propositions (from a set of atomic propositions AP). The transition probability matrix defines $\boldsymbol{P}(s,s')$ the probability of the next state s' based on the value of the current state s. It is subject to $\sum_{s' \in S} \boldsymbol{P}(s,s') = 1$ for all $s \in S$, i.e. a next state is chosen with probability 1. We say that a transition between two states $s, s' \in S$ exists if $\boldsymbol{P}(s,s') > 0$.*

Intuitively, a DTMC D is a set of states corresponding to possible configurations of the system that is modelled and with transitions between states occurring in discrete time-steps. Each transition is augmented with the probability of that transition being taken. The labelling

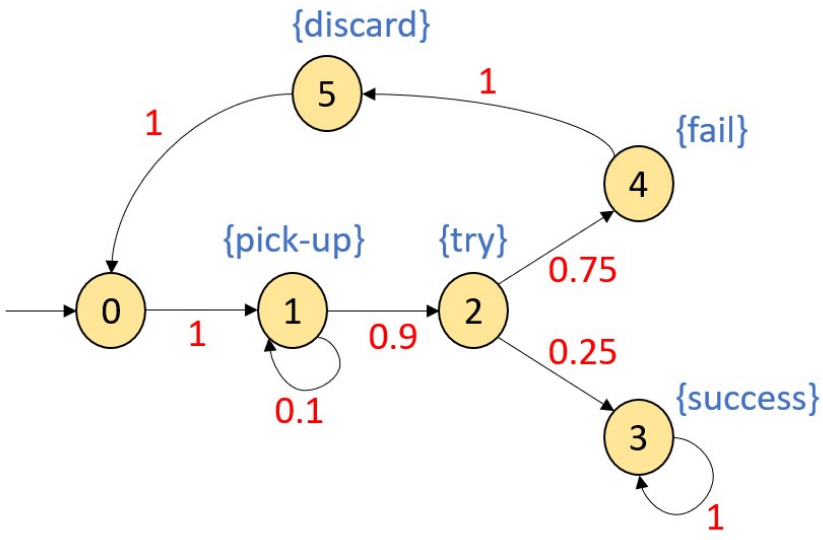


Figure 3.1: An example of a Discrete Time Markov Chain.

function is often omitted from the tuple if there are no properties of interest with which to label any of the states.

Example 1. *Figure 3.1 is a diagram of a Discrete Time Markov Chain with* $S = \{0,1,2,3,4,5\}$ *and initial state* $s_{init} = 0$*. The corresponding probability matrix is*

$$\boldsymbol{P} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0.1 & 0.9 & 0 & 0 & 0 \\ 0 & 0 & 0.25 & 0.75 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

with $\boldsymbol{P}(i,j)$ *denoting the probability to transition from state* i *to state* j*.*

It is often necessary to consider the underlying graph of a DTMC as graph theory can provide insight into the system being analysed.

Definition 1.1 The underlying graph of a DTMC D is a graph $G = (V,E)$, where $V = S$ is a set of vertices corresponding to the states of the model, and $E = \{(s,s')$ such that $\mathbf{P}(s,s') > 0\} \subseteq V \times V$ are directed edges connecting vertices of states between which there is a positive transition probability.

We will introduce some additional frequently used terminology and point out some of the assumptions that are being made:

- The Markov property, often referred to as memorylessness, that for a given current state,

future states are independent of past states, is formally defined as:

$$Pr(X_{t+1}|X_t, X_{t-1}, ..., X_0) = Pr(X_{t+1}|X_t)$$

- **P** is called a *stochastic* matrix as it satisfies:
  $\mathbf{P}(s,s') \in [0,1]$ for all $s,s' \in S$ and $\sum_{s' \in S} \mathbf{P}(s,s') = 1$ for all $s \in S$.
- A state s for which $P(s,s) = 1$ and consequently $\mathbf{P}(s,s') = 0$ for all $s' \neq s$ is called an *absorbing state*.
- While S can be any countable set, for the rest of this thesis we will assume that the set of states is finite.
- The initial state s_{init} can be generalised to an initial *probability distribution* of states $s_{init} : S \to [0,1]$.
- Transition probabilities are defined to be real values, but are often restricted to rational values for software purposes.
- A state s' is reachable from s if there is a finite path from s to s'.
- A subset of states is *strongly connected* if any two states in it are connected by a finite path passing only though states in the subset.
- A maximally strongly connected set of states is a strongly connected set of states for which no superset is strongly connected, and is called a *strongly connected component (SCC)*.
- A *bottom strongly connected component (BSCC)* is a strongly connected component for which no state outside of it is reachable from it.

A path ω in a Discrete Time Markov Chain is a infinite sequence of states $s_0 s_1 s_2 s_3 ...$ such that $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. Such paths represent different executions or different possible behaviours of the underlying system. For example, the DTMC in Figure 3.1 has 012(3), 012450112(3), and (01245) as possible paths. It is sometimes helpful to consider paths of finite length, which can be obtained by restricting an infinite path to a given length.

Definition 2. *The set of all infinite paths that start from a state s is denoted by* $Path(s)$. *Similarly,* $Path_{fin}(s)$ *denotes the set of all finite paths starting from s.*

The above definition can be simplified to $Path$ and $Path_{fin}$ if the state s is the initial state of the model. For finite paths ω we can define the cylinder set $Cyl(\omega)$ as the set of all infinite paths that share ω as their starting prefix. For example. the DTMC on Figure 3.1 has the cylinder set $Cyl(0124)$ as all executions that directly perform a try and that first try results in a failure. The probability of a finite path $s_0 s_1 s_2 s_3 ... s_n$ is given by $\prod_{i=0}^{n-1} \mathbf{P}(s_i, s_{i+1})$.

It is often useful to determine the probability that a system is in a given state after specified number of steps have been taken from a particular initial state.

Definition 3. *The probability that a system is in state s' after k steps have occurred starting from an initial state s is called the transient probability and is given by*

$$\pi_{s,k}(s') = \sum_{s'' \in S} \boldsymbol{P}(s'', s') \cdot \pi_{s,k-1}(s'')$$

$$\pi_{s,0}(s') = \begin{cases} 1 & s' = s \\ 0 & s' \neq s \end{cases}.$$

This is a recursive definition based on the transient probability at the previous step and the probabilities of the incoming transitions. The transient probabilities at step 0 are based on the initial state (or initial probability distribution). It is often useful to combine the transient probabilities into a vector.

Definition 3.1 The transient state distribution is the discrete probability distribution $\underline{\pi}_{s,k} : S \to [0,1]$ where

$$\underline{\pi}_{s,k} = \pi_{s,k}(s') \text{ for all } s' \in S.$$

The recursive relation above can then be rewritten as

$$\underline{\pi}_{s,k} = \underline{\pi}_{s,k-1} \cdot \mathbf{P} = \underline{\pi}_{s,0} \cdot \mathbf{P}^k,$$

so the transient state distribution can be calculated through successive matrix multiplications. We can view $\mathbf{P}^k$ as transition probabilities for performing k steps at once, while $\mathbf{P}$ for performing a single step.

In certain cases transient probabilities are not very useful, it is often more helpful to study the long-run behaviour of the system.

Definition 4. *The limiting distribution, if it exists, is the limit*

$$\underline{\pi}_s = \lim_{k \to \infty} \underline{\pi}_{s,k}.$$

This is used to give a measure of the percentage of time spent by the system in each state in the long run. The underlying graph must be considered in order to determine the existence of this limit.

If all states of a Markov chain belong to a single BSCC, then the Markov chain is *irreducible*, and reducible otherwise. The limiting distribution for a finite irreducible DTMC always exists and is not dependent on the initial state (initial distribution). In this case the limiting distribution

is known as steady-state probabilities and can be computed by solving the balance equations

$$\underline{\pi} \cdot \mathbf{P} = \underline{\pi} \text{ and } \sum_{s \in S} \underline{\pi}(s) = 1.$$

3.2 Property specification

In order for model checkers to perform formal verification of properties of the system that has been modelled, these properties need to be specified using a formal language. Such a formal language that can be used to specify the changes in the behaviour of a system over time is called a *temporal logic*. Temporal logic extends propositional logic with temporal operators.

Definition 5. *A labelled state-transition system (LTS) is a tuple $(S, s_{init}, \rightarrow, L)$, where: S is a set of states called the state space, $s_{init} \in S$ is the initial state (often labelled s_0), $\rightarrow \subseteq S \times S$ is a transition relation, and $L : S \rightarrow 2^{AP}$ is a function labelling states with Atomic Propositions (from a set of atomic propositions AP). A DTMC $(S, s_{init}, \mathbf{P}, L)$ has a corresponding LTS $(S, s_{init}, \rightarrow, L)$, where $\rightarrow = \{(s, s') \text{ such that } \mathbf{P}(s, s') > 0\}$.*

A well-known temporal logic used for non-probabilistic models is Computation Tree Logic (CTL) [43]. This logic uses a *b*ranching notion of time: at any time step system behaviour can move to a different path as a result of an event (rather than a single path being executed from an initial state). CTL formulae are interpreted based on an LTSs and provide a way to combine atomic propositions to make statements about the states of the system. Formulas are constructed from atomic propositions using temporal and logical operators.

Definition 6. *Computation Tree Logic has the following syntax:*

$$\Phi = true \mid false \mid a \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg\Phi \mid \Phi \Rightarrow \Phi \mid \Phi \Leftrightarrow \Phi$$

$$\mid \quad AX\ \Phi \mid EX\ \Phi \mid AF\ \Phi \mid EF\ \Phi \mid AG\ \Phi \mid EG\ \Phi \mid A\ [\Phi \cup \Phi] \mid E\ [\Phi \cup \Phi]$$

where a is an atomic proposition, and A, E, X, G, F, $\cup$ are the temporal operators "All", "Exists", "Next", "Globally", "Finally", and "Until" respectively.

Probabilistic Computation Tree Logic (PCTL) [107] is a probabilistic extension of the non-probabilistic CTL that adds a probabilistic operator P to the syntax of CTL. This enables the verification of properties that are probabilistically quantified, such as soft deadline properties, e.g. "after a button is pressed, there is at least a 99% probability that the device will shut down within 2 seconds". Probabilistic model checkers widely use PCTL as a property specification language.

Definition 7. *Probabilistic Computation Tree Logic has the following syntax:*

$$\Phi = true \mid a \mid \Phi \wedge \Phi \mid \neg\Phi \mid P_{\sim p}[\Psi]$$

$$\Psi = X\Phi \mid \Phi \cup^{\leq k} \Phi \mid \Phi \cup \Phi$$

where a is an atomic proposition, $p \in [0,1]$ *is a probability bound, and* $\sim \in \{<, >, \leq, \geq\}, k \in \mathbb{N}$.

3.3 Markov Decision Processes

The DTMCs described in Section 3.1 offer a purely probabilistic way of modelling systems. Successor states are chosen based on predefined probabilities for each state. However, it is often desirable to also represent nondeterminism, i.e. decisions undertaken by the system. This nondeterminism can be in either a controllable setting (e.g., decisions made by a player in game theory) or an uncontrollable setting (e.g., the interweaving of processes in a concurrent system).

Definition 8. *A (labelled) MDP* $\mathscr{M}$ *is a tuple*

$$\mathscr{M} = (S, s_0, \mathbf{Steps}, L)$$

where: S is a countable set of states called the state space, $s_0 \in S$ *is the initial state,* $\mathbf{Steps} : S \to 2^{Act \times Dist(S)}$ *is the state transition probability function where Act is a set of actions and* $Dist(S)$ *is the set of discrete probability distributions over the set S, and* $L : S \to 2^{AP}$ *is the labelling function assigning atomic propositions to the set of states.*

We denote the set of actions enabled in a state $s \in S$ *by* $Act(s)$*, and the set of states reachable from that state by* $\mathbf{Steps}(s)$*. We assume that* $Act(s) \neq \emptyset$ *and consequently* $\mathbf{Steps}(s) \neq \emptyset$*, i.e. there are no deadlocks in the process.*

MDPs often model systems which exhibit both nondeterministic and probabilistic behaviour (for example, the asynchronous operation of multiple probabilistic processes in a randomised distributed algorithm). Intuitively, in each state $s \in S$ there is a nondeterministic choice between the elements of $Act(s)$. Then, there is a probabilistic choice according to the distribution $\mu \in Dist(s)$ related to that action to select the next state of the model. Thus, a path through a DTMC is resolved through a series of nondeterministic and probabilistic choices.

Analogously to the probability matrix of a DTMC, it is often useful to view the transition probability function as a matrix. This is a rectangular matrix with a number of columns equal to the number of states $|S|$, and number of rows equal to the total number of distributions of the transition probability function $\sum_{s \in S} |\mathbf{Steps}(s)|$.

Example 2. *Figure 3.2 is a diagram of a Markov Decision Process with* $S = \{0,1,2,3,4\}$ *and*

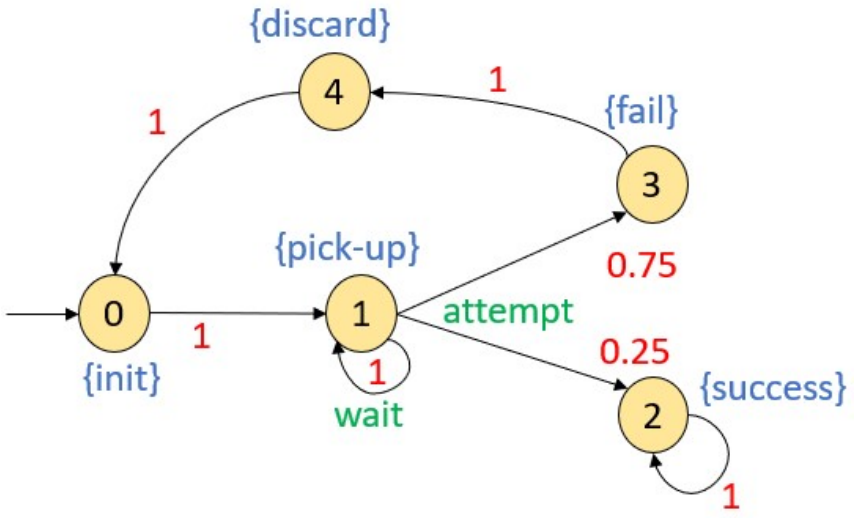


Figure 3.2: An example of a Markov Decision Process.

initial state $s_{init} = 0$. *The corresponding transition probability function is*

$$\textbf{\textit{Steps}} = \left(\begin{array}{cccccc} 0 & 1 & 0 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.25 & 0.75 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \end{array}\right)$$

where the rows of **Steps** *denote different probability distributions. For clarity, distributions associated with different states are separated with horizontal lines. Each column of the matrix represents the probability that the system moves to the state associated with it. For example,* $\textbf{\textit{Steps}}(3,2)$ *signifies that if the "attempt" action is taken in state 1, there is a 0.25 probability to transition to state 2.*

A path ω in an MDP is a infinite sequence of states and distribution pairs $s_0\mu_0 s_1\mu_1 s_2\mu_2 s_3\mu_3\ldots$ such that $\mu_i \in \textbf{Steps}(s_i)$ and $\mu_i(s_{i+1}) > 0$ for all $i \geq 0$. As with DTMCs, these paths also represent different executions or different possible behaviours of the underlying system. For example, the MDP in Figure 3.2 has 01(2), 01234011(2), and (01234) as possible paths (distributions omitted for clarity). Paths of finite length can be obtained by restricting an infinite path to a given length. $Path(s)$ and $Path_{fin}(s)$ are defined in exactly the same way as their equivalents for DTMCs from Definition 2.

To reason about the behaviour of an MDP it is necessary to first resolve all nondeterministic choices. Doing so reduces an MDP to a DTMC, for which we know how to compute probabilities. An *adversary* (also known as "strategy", "policy" or "scheduler") is used to resolve nondeterministic choices in an MDP.

Definition 9. *An adversary σ of an MDP $\mathscr{M}$ is a function mapping every finite path $\omega = s_0(a_0,\mu_0)s_1\ldots s_n$ to an element $\sigma(\omega)$ of* **Steps**(s_n).

The set of all adversaries is denoted by Adv_M or by Adv if it is clear which MDP it is associated with.

We give an example of adversaries in the context of the MDP of Example 2. Note that in that MDP, the only state involving a nondeterministic choice is state 1, i.e. $s = 1$ is the only $s \in S$ for which $|\mathbf{Steps}(s)| > 1$. Therefore, this is the only state for which the adversary needs to resolve the nondeterminism. Let μ_0 be the probability distribution associated with the "wait" choice and μ_1 the one associated with the "attempt" choice. An adversary σ_1 can always choose the "wait" action:

$$\sigma_1(01) = (wait, \mu_0)$$

$$\sigma_1(011) = (wait, \mu_0)$$

$$\sigma_1(0111) = (wait, \mu_0), etc$$

while an adversary σ_2 can choose the "wait" action once and choose "attempt" afterwards:

$$\sigma_2(01) = (wait, \mu_0)$$

$$\sigma_2(011) = (attempt, \mu_1)$$

$$\sigma_2(0113401) = (attempt, \mu_1), etc$$

An adversary can be used to define a subset of the set of all paths $Path^{\sigma_1}(s) \subseteq Path(s)$ to be the set of all infinite paths starting from state s where nondeterminism is resolved by σ_1. Similarly an adversary σ can be used to induce an infinite-state DTMC D^σ from the MDP it is associated with.

Definition 10. *An induced DTMC D^σ is a DTMC $D^\sigma = (Path^\sigma(s), s, \mathbf{P}^\sigma_s)$ where the states of the DTMC are the finite paths obtained by resolving nondeterminism using σ starting from state s; the initial state is state s; and*

$$\mathbf{P}^\sigma_s(\omega,\omega') = \mu(s') \text{ if } \omega' = \omega(\mu)s' \text{ and } \sigma(\omega) = \mu$$

$$\mathbf{P}^\sigma_s(\omega,\omega') = 0 \text{ otherwise}$$

3.4 Costs and Rewards

DTMCs can be augmented by a cost or reward structure. This is done by annotating certain states and/or transitions with real valued variables, which accumulate whenever the state/transition is passed during execution. These values can be used to represent a variety of useful properties

of the system, such as elapsed execution time, number of tasks performed, length of a queue, energy consumption, etc. Mathematically, rewards and costs are the same thing but the typical convention is that costs need to be minimised while rewards should be maximised. This reward structure enables a variety of quantitative analysis to be performed on the system through the use of reward-based properties. Such properties can be specified through the use of an appropriate specification language; for example, PRISM uses an extension of PCTL.

Definition 11. *A reward structure is a pair* $(\underline{\rho}, \iota)$ *that is associated with a DTMC* $(S, s_{init}, \mathbf{P}, L)$*, where*

- $\underline{\rho} : S \to \mathbb{R}_{\geq 0}$ *is the state reward function, and*
- $\iota : S \times S \to \mathbb{R}_{\geq 0}$ *is the transition reward function.*

This reward structure can be used to reason about the expected value of the rewards at a given point of the operation of the system being modelled. This can be achieved in two ways: instantaneous, where the property is concerned with the expected value of the state reward function at a given time k, or cumulative, where the property investigates the expected sum of the rewards of both the state and transition reward functions before a desired point of operation of the system. The latter case can either be time-bounded, so we look at the expected reward accumulated up until a time k, or reachability-bounded, so we look at the expected reward accumulated until one of some desired states is reached.

3.5 Tools for probabilistic model checking

Probabilistic model checkers are tools for the formal modelling and analysis of systems which exhibit probabilistic or random behaviour. These tools accept models encoded in an appropriate formal language as input and a number of desirable properties expressed in a suitable logic.

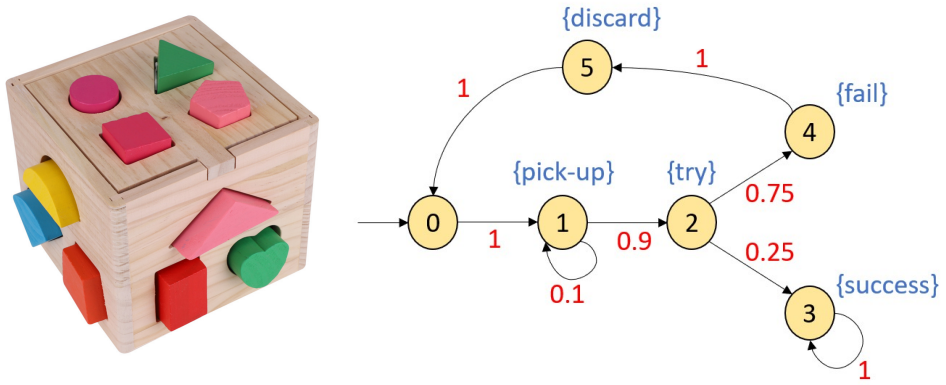
3.5.1 The PRISM Model Checker

PRISM is a symbolic probabilistic model checker that enables formal modelling and analysis of systems with probabilistic or random behaviour.

Types of Prism models

PRISM provides support for a number of Markov chain variants including:

1. Discrete Time Markov Chains (DTMCs). Properties can be expressed in PCTL and LTL (Linear Temporal Logic).
2. Continuous-Time Markov Chains (CTMCs). Properties are expressed in CSL (Continuous Stochastic Logic), an extension of PCTL for CTMCs.



(a) Toy game of matching shapes to holes.

(b) State diagram of a robot trying to solve the game.

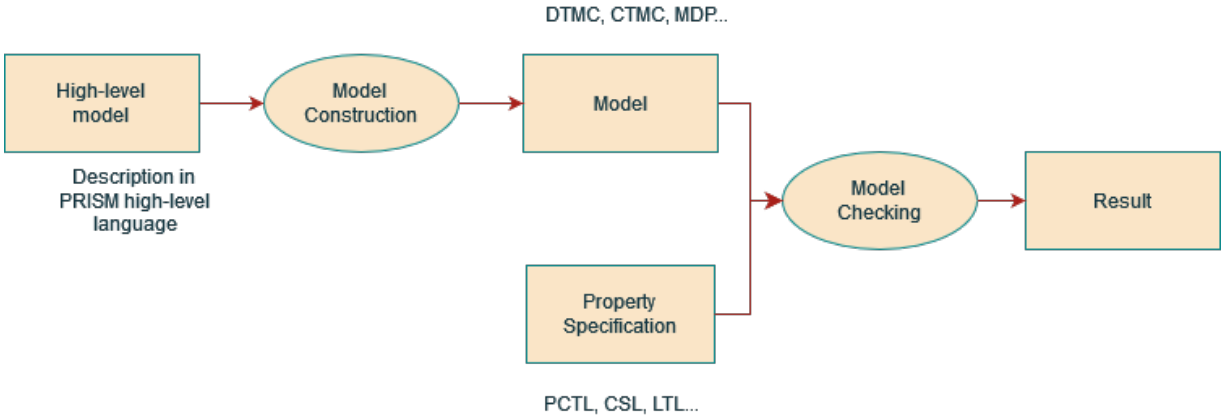


Figure 3.4: An overview of the workflow of model checking in PRISM.

3. Markov Decision Processes (MDPs). Properties are expressed in PCTL* and LTL.

4. Probabilistic Timed Automaton (PTA). Uses PCTL.

The PRISM model checker provides a high-level modelling language which can be used for a wide range of model analysis techniques. This language is a simple state-based language based on the Reactive Modules formalism [7]. The model checking process is split into two distinct phases: model construction and model checking. This is illustrated in Figure 3.4: First the user-provided specification of the system in the PRISM modelling language is used to construct a model for the system (e.g. DTMC, CTMC, MDP, etc). This model is then used in conjunction with the property specifications provided by the user in the corresponding probabilistic temporal logic in order to perform the model checking process.

PRISM modelling language

The core concept of the PRISM modelling language is to express the components of the system being modelled as separate modules. The state of each component is then recorded using variables, either local or global, and the overall system behaviour is expressed through the mod-

ules and their interactions. Variables in PRISM are finite valued and can be either booleans or have integer ranges. Updates to the states of each module are defined using PRISM's guarded commands which have the following form:

[label] guard -> prob_1 : update_1 + ... + prob_n : update_n;

where:

- `label` is a synchronisation flag for transitions that need to be performed simultaneously. This label is an optional argument;
- `guard` is a predicate that specifies the state(s) to which the transition can be applied;
- `prob_1, ..., prob_n` are the probabilities of updates being performed. For each command these probabilities must sum up to 1.0;
- `update_1, ..., update_n` are the possible updates performed by this transition. Each update must be a conjunction of statements of the form `(var' = new_value)`, where `var'` indicates the new value of the variable `var`.

Listing 3.1 shows an example PRISM specification corresponding to the Discrete Time Markov chain from the example on Figure 3.1. There is a single module called "main" with one local variable that keeps track of the state that the module is in. The module has six states and six transition statements (which define eight transitions as lines 10 and 12 result in two transitions each).

```
1  // example sorting model
2  dtmc
3  const int STATES = 5;
4
5  module main
6  state: [0..STATES] init 0; // the possible states that the decision process
       goes through,
7
8  [] state = 0 -> 1.0: (state'= 1);
9  // pick-up
10 [] state = 1 -> 0.9: (state'=2) + 0.1: (state'=1);
11 // try
12 [] state = 2 -> 0.25: (state'=3) + 0.75: (state'=4);
13 // success
14 [] state = 3 -> (state'= 3);
15 // fail
16 [] state = 4 -> (state'= 5);
17 // discard
18 [] state = 5 -> (state'= 0);
19
```

```
20 endmodule
```

Listing 3.1: Example PRISM model: sorting

Verification of properties using PRISM

Verification in PRISM is achieved using properties described in the PRISM property specification language, which includes various logics such as PCTL, (probabilistic) LTL, CSL, CTL and PCTL*. The choice of which is used depends on the model type. We focused only on a small set of properties in this thesis, so we do not describe all of these logics in full. However we present enough detail to show how PRISM properties that we use are constructed.

PRISM properties have the general form:

```
operator bound [ pathprop ]
```

where `operator` is one of the probabilistic operators **P**, **S**, and **R** or the non-probabilistic operators **A** and **E**, as well as any temporal operators (e.g. eventually **F** and globally **G**). Here, `pathprop` can be any PRISM expression with the optional inclusion of any of the aforementioned operators.

An example property of interest for the DTMC in Figure 3.1 might be

```
P=?  [F<10 s=3]
```

Which is equivalent to "What is the probability that within 10 units of time state three is reached?". The **P** operator is applicable to all types of models supported by PRISM and it used to analyse the probability of an event's occurrence. For verification of models that include non-determinism (e.g. MDPs), we are unable to reason about probabilities of events in the same way as the probabilities would change depending on how the non-determinism is resolved. Therefore, for models involving non-determinism, we instead use the **P**max and **P**min operators to inquire about the maximal and minimal probabilities over the set of all adversaries.

Another probabilistic operator of interest is the **S** operator, which is used to reason about the steady-state behaviour of a model, the behaviour that the model reaches in its equilibrium. PRISM currently supports this operator for DTMCs and CTMCs. For example, the property

```
S<0.05 [ slot = 1 ]
```

means that "the long-run probability of the slot variable being equal to one is less than 5%". Steady-state properties have been shown to be well-defined for finite DTMCs and CTMCs [220].

3.6 State space reduction techniques

When model checking a complex system hardware and protocol verification, concurrent and real-time systems, and hybrid systems, the state space explosion problem is often encountered. This is due to the fact that the number of states grows exponentially in the number of components in a system, leading to a corresponding increase in the time and resources required for verification. State space reduction techniques have been developed with the aim to help address this challenge.

State space reduction techniques aim to minimise the number of states that need to be considered while preserving the important aspects of the system's behaviour. The goal is to obtain an abstract representation of the system that captures its essential characteristics while reducing the complexity of the analysis.

There are several techniques used to achieve state space reduction, including abstraction, symmetry reduction, partial order reduction, and compositional verification. Abstraction involves replacing a detailed model of a system with a simplified model that only captures the system's behaviour that is of interest. Symmetry reduction takes advantage of the symmetries in a system to reduce the number of states that need to be considered. Partial order reduction exploits the independence of events to reduce the number of states that need to be explored. Compositional verification involves breaking the system into smaller parts and analysing them independently before combining them to analyse the system as a whole.

3.6.1 Symmetry reduction, Automorphisms and Quotient MDPs

Symmetry reduction techniques are methods used in model checking to reduce the size of a model by exploiting its symmetries. Symmetry refers to the property of a system where some parts of it are indistinguishable from each other with respect to some property. For example, in a model of a distributed system, the processes may be identical in behaviour, so any permutation of the processes produces an equivalent system.

Symmetry reduction techniques take advantage of such symmetries to reduce the number of states that need to be analysed during model checking. The basic idea is to identify symmetries in the system and then partition the state space of the model into equivalence classes, each of which contains states that are symmetrically equivalent. The model checker then only needs to explore one representative state from each equivalence class, rather than exploring all of them, thereby significantly reducing the state space and the computational resources required for model checking.

To leverage the symmetry present in an MDP we need to be able to reason about the underlying properties and behaviour of different MDPs.

Definition 12. *Let* $\mathscr{M} = (S, s_0, \mathbf{Steps})$ *and* $\mathscr{M}' = (S', s_0', \mathbf{Steps}')$ *be two MDPs, and* $\alpha : S \to S'$ *be a bijection between their state spaces such that* $\alpha(s_0) = s_0'$*. Suppose that for* $s \in S$ *we have*

$\mu \in \mathbf{Steps}(s)$ if and only if there exists $\mu' \in \mathbf{Steps}'(\alpha(s))$ such that for all $t \in S$, $\mu(t) = \mu'(\alpha(t))$. We call α an isomorphism from M to M', and the two MDPs are said to be isomorphic.

As is typical in Group Theory, we now apply this concept to a single MDP in order to analyse the effects its symmetries have on it.

Definition 13. *An isomorphism from an MDP $\mathscr{M}$ to itself is called an automorphism. The set of all isomorphisms from $\mathscr{M}$ to $\mathscr{M}$ constitutes a group under function composition. This group is called the automorphism group of M and is denoted $Aut(M)$.*

We then use the automorphism group to establish which states and transitions of the MDP are identical under symmetry.

Definition 14. *Let G be a subgroup of the automorphism group of an MDP $\mathscr{M} = (S, s_0, \mathbf{Steps})$, $G \leq Aut(\mathscr{M})$. The orbit relation $\sim$ is defined for all $s,t \in S$ so that $s \sim t$ if and only if there exists $g \in G$ such that $t = g * s$ where $*$ denotes group action.*

The operation used is group action, so it follows that $\sim$ is an equivalence relation. The *orbit* of s is denoted $Orb(s)_G$ and is the equivalence class of s under $\sim$, i.e. the set $[s]_G = t : t \sim s$. We write $Orb(x)$ and $[s]$ if the group G is clear from the context.

We use the orbit relation to construct a version of an MDP which abstracts away the symmetry of the original.

Definition 15. *Let there exist an ordering for the state space S of an MDP $\mathscr{M}$, and let the smallest element of $Orb(s)$ be denoted $\min[s]$ for every state $s \in S$. Then, for any $G \leq Aut(\mathscr{M})$, the quotient MDP with respect to G for an MDP $\mathscr{M} = (S, s_0, \mathbf{Steps})$ is defined as the MDP $\overline{\mathscr{M}} = (\overline{S}, \overline{s_0}, \overline{\mathbf{Steps}})$ where:*

- $\overline{S} = \{\min[s] : s \in S\}$
- $\overline{s_0} = \min[s_0]$
- $\overline{\mathbf{Steps}}$ *is constructed such that for each* $\min[s] \in \overline{S}$ *and* $\mu \in \mathbf{Steps}(\min[s]), \overline{\mathbf{Steps}}(\min[s])$ *contains a distribution* $\overline{\mu}$ *where for each* $\min[t] \in \overline{S}, \overline{\mu}(\min[t]) = \sum_{u \in [t]} \mu(u)$.

The element $\min[s]$ is called a unique representative of $[s]$. In many cases analysis and verification of the full system can instead be performed on the (smaller) quotient MDP. The correspondence between PCTL properties of isomorphic MDPs under an appropriate substitution of atomic properties is established by the following theorem (proved in [171]). Note that if s is a state of MDP $\mathscr{M}$ and $a \in A$ is an atomic proposition, then $\mathscr{M}, s \models a$ if a evaluates to true at s.

Theorem 1. *Let $\mathscr{M} = (S, s_0, \mathbf{Steps})$ and $\mathscr{M}' = (S', s_0', \mathbf{Steps}')$ be two isomorphic MDPs, A and A' the corresponding sets of atomic propositions, $\gamma : A \to A'$ a bijection, and δ and isomorphism*

from $\mathscr{M}$ to $\mathscr{M}'$, such that for every $s \in S$ and $a \in A$, $\mathscr{M}, s \models a \Leftrightarrow \mathscr{M}', \delta(s) \models \gamma(a)$. Then for any PCTL formula Φ over A and $s \in S$,

$$\mathscr{M}, s \models \Phi \Leftrightarrow \mathscr{M}', \delta(s) \models \gamma(\Phi)$$

where $\gamma(\Phi)$ is the PCTL formula obtained by replacing every atomic proposition a in Φ with $\gamma(\Phi)$.

For a PCTL formula Φ and $G \leq Aut(\mathscr{M})$, we say that Φ is symmetric with respect to G if for every maximal propositional sub-formula f that appears in Φ and for any state $s \in S$, we have $\mathscr{M}, s \models f \Rightarrow \wedge_{s' \in [s]} \mathscr{M}, s \models f$ [69].

If we can check a PCTL formula that is symmetric with respect to a group of MDP automorphisms then we can check whether it holds by considering the quotient MDP instead (which may be considerably smaller). The following theorem is proved in [171].

Theorem 2. *Let f be a PCTL formula which is symmetric with respect to $G \leq Aut(\mathscr{M})$. Let $\overline{\mathscr{M}}$ be the quotient MDP for $\mathscr{M}$ with respect to G. Then $\mathscr{M} \models \Phi \Leftrightarrow \overline{\mathscr{M}} \models \Phi$.*

The size of $[s]$ is bounded by $\mid G \mid$, hence the minimum size of $\overline{S}$ is $\mid S \mid / \mid G \mid$. For highly symmetric systems with n components, it is possible that $\mid G \mid = n!$, so verifying a PCTL property against the quotient MDP can potentially offer a significant reduction in resource requirements.

3.6.2 Counter abstraction/ Generic representatives

The construction of the orbit relation for symbolic model checking can be a resource intensive process. Using *generic representatives* allows symmetry reduction to be applied without the construction of the orbit relation. This is achieved via a source-to-source translation at the language level [58] and without any changes to the existing model checking algorithm.

We will explain the idea using an example. Consider a communication protocol establishment algorithm for four identical devices. Each device has three possible local states *disconnected* (D), *attempting* (A), and *connected* (C). The global states (D, D, D, A), (D, D, A, D), (D, A, D, D) and (A, D, D, D) are symmetrically equivalent as each of them has two out of the three processes in the *disconnected* state and the last one in the *attempting* state. A generic representative only keeps track of *the number of* processes in each local state, discarding any information about the state of individual processes. The global states listed above thus all have the single generic representative (3D, 1A, 0C), which has multiple instances of the identical processes substituted with *counters*. A counter variable is created for each local state of the identical process with that variable keeping a record of the number of processes currently in its corresponding local state. When stating a generic representative, we can omit the counters with zero values, so the generic representative above can be written as (3D, 1A).

3.6.3 Symmetry reduction in PRISM

There are two PRISM-related tools capable of performing state space reduction. Both of them use symmetry to accomplish this goal.

PRISM-symm

PRISM-symm [144] is based on an efficient algorithm for the construction of quotient models. It uses a symbolic implementation based on the multi-terminal binary decision diagrams data structure [14] [83]. The idea is to create an MTBDD representing the quotient model directly from a high-level model description. However, this was impossible due to an implicit introduction of inter-component dependencies during symmetry reduction. Thus, PRISM-symm's approach first constructs a symbolic representation for the full model and then reduces it to one representing the quotient model. In explicit model checking this would invalidate the symmetry reduction being performed, as the construction and storage of the full model is the main bottleneck of the process; however, this is not the case in a symbolic setting. There it is often the case that an MTBDD representing a full model can be accomplished (even for very large models), but the model checking of that MTBDD cannot be performed afterwards. In such cases the symmetry reduction of PRISM-symm has the potential to be very useful.

The paper that introduces PRISM-symm [144] uses four case studies for its evaluation: a BitTorrent peer-to-peer protocol, the randomised Byzantine agreement protocol, the shared coin randomised consensus protocol, and the IEEE 802.3 CSMA/CD communication protocol. The results show a significant increase in the size of the models that can be modelled and verified.

GRIP

GRIP (Generic representatives in PRISM) is based on the generic representatives approach and similarly to PRISM-symm aims to overcome the problem of combining symbolic state space representation with symmetry reduction. GRIP does so by applying counter abstraction directly to the model specification, prior to the construction of any MTBDD. This circumvents the need to first construct the full, unreduced model. Furthermore, as GRIP only acts upon PRISM specifications (i.e. as a pre-processor), it has the benefit of also being applicable to any tool that uses the PRISM modelling language (e.g. the statistical model checker Ymer [244]), as well as any input language that a PRISM specification can be translated to (e.g. the Markov Reward Model Checker MRMC [127]).

We present the translation rules used by GRIP, introduced in [58], in Figure 3.5 and briefly explain them.

An SPSL specification contains a number of module declarations `module`, each defining a family of symmetric modules, in addition to a set of optional global variable declarations. A *module* declaration consists of a name, a set of local variable declarations, and a set of transition

$\mathscr{P}$	$\mathfrak{h}(\mathscr{P})$
global-variables module ... module	global-variables $\mathfrak{h}$(module) ... $\mathfrak{h}$(module)
module	$\mathfrak{h}$(module), with $m = init(M)$ and $t = \|S(M)\|$

`module` $M[k]$ {
var-decl*
statement(M) ... statement(M)
}
	`module generic_`$M[1]$ {
	`count_`M`_1 : [0..`k`] init 0`
	... `count_`M`_`$f_M(m)$ `: [0..`k`] init` k ...
	`count_`M`_`t `: [0..`k`] init 0`
	$\mathfrak{h}$(statement(M)) ... $\mathfrak{h}$(statement(M))
	}

statement(M), where e is local-expr(M)	$\mathfrak{h}$(statement(M)), with $\mathsf{SAT}_M(e) = \{l^1, \ldots, l^z\}$
$e_i \wedge$ expr(M_i) $\rightarrow$ stoch-update(M)	`count_`M`_`$f_M(l^1) > 0 \wedge \mathfrak{h}$(expr$(M_i), l^1)$ $\rightarrow \mathfrak{h}$(stoch-update$(M), l^1)$ ... `count_`M`_`$f_M(l^z) > 0 \wedge \mathfrak{h}$(expr$(M_i), l^z)$ $\rightarrow \mathfrak{h}$(stoch-update$(M), l^z)$
stoch-update(M)	$\mathfrak{h}$(stoch-update$(M), l)$
expr(M_i):update$(M) + \ldots$ $+$expr(M_i):update(M)	$\mathfrak{h}$(expr$(M_i), l)$:$\mathfrak{h}$(update$(M), l) + \ldots$ $+\mathfrak{h}$(expr$(M_i), l)$:$\mathfrak{h}$(update$(M), l)$
update(M), where $v^j \in var(M)$, $g^j \in global$ and e^j/d^j has form local-expr(M)/expr(M_i)	$\mathfrak{h}$(update$(M), l)$, where $l' = l[v^1 := eval(l, e^1), \ldots, v^t := eval(l, e^t)]$
`skip` $(v_i^1 := e_i^1) \| \ldots \| (v_i^t := e_i^t)$ $\|(g^1 := d^1) \| \ldots \| (g^r := d^r)$	`skip` (`count_`M`_`$f_M(l)$ `:=count_`M`_`$f_M(l) - 1$) $\|$(`count_`M`_`$f_M(l')$ `:=count_`M`_`$f_M(l') + 1$) $\|(g^1 := \mathfrak{h}(d^1, l)) \| \ldots \| (g^r := \mathfrak{h}(d^r, 1))$
expr(M_i), where e has form local-expr(M)	$\mathfrak{h}$(expr$(M_i), l)$
e_i	$eval(l, e)$
symm-expr	$\mathfrak{h}$(symm-expr)
$\sum_{1 \le j \ne i \le \#M} e_j$	$\sum_{m \in S(M)} (eval(m, e) *$ `count_`M`_`$f_M(m)) - eval(l, e)$
$\prod_{1 \le j \ne i \le \#M} e_j$	$\prod_{m \in S(M)} (eval(m, e)$**`count_`$M$`_`$f_M(m)) / eval(l, e)$
$\bigwedge_{1 \le j \ne i \le \#M} e_j$	$\sum_{m \in \mathsf{SAT}_M(e)}$ `count_`M`_`$f_M(m) = \#M$ (if $l \models e$)
	$\sum_{m \in \mathsf{SAT}_M(e)}$ `count_`M`_`$f_M(m) = \#M - 1$ (if $l \not\models e$)
$\bigvee_{1 \le j \ne i \le \#M} e_j$	$\sum_{m \in \mathsf{SAT}_M(e)}$ `count_`M`_`$f_M(m) > 0$ (if $l \not\models e$)
	$\sum_{m \in \mathsf{SAT}_M(e)}$ `count_`M`_`$f_M(m) > 1$ (if $l \models e$)
expr$(M_i) \bowtie$ expr(M_i)	$\mathfrak{h}$(expr$(M_i), l) \bowtie \mathfrak{h}$(expr$(M_i), l)$
$\neg$expr(M_i)	$\neg \mathfrak{h}$(expr$(M_i), l)$
(expr(M_i))	($\mathfrak{h}$(expr$(M_i), l)$)
symm-expr, where e has form local-expr(N)	$\mathfrak{h}$(symm-expr)
constant	constant
name (where name is a global variable)	name
$\sum_{1 \le j \le \#N} e_j$	$\sum_{l \in S(N)} (eval(l, e) *$ `count_`N`_`$f_N(l))$
$\prod_{1 \le j \le \#N} e_j$	$\prod_{l \in S(N)} (eval(l, e)$**`count_`$N$`_`$f_N(l))$
$\bigwedge_{1 \le j \le \#N} e_j$	$\sum_{l \in \mathsf{SAT}_N(e)}$ `count_`N`_`$f_N(l) = \#N$
$\bigvee_{1 \le j \le \#N} e_j$	$\sum_{l \in \mathsf{SAT}_N(e)}$ `count_`N`_`$f_N(l) > 0$
symm-expr $\bowtie$ symm-expr	$\mathfrak{h}$(symm-expr) $\bowtie \mathfrak{h}$(symm-expr)
$\neg$symm-expr	$\neg \mathfrak{h}$(symm-expr)
(symm-expr)	($\mathfrak{h}$(symm-expr))

Figure 3.5: Rules for translating an SPSL specification $\mathscr{P}$ to a generic form $\mathfrak{h}(\mathscr{P})$. Taken from [58].

commands (statements). The number of copies of each symmetric module is specified via the use of PRISM's module renamings. Without loss of generality we assume that all variable and module names are distinct. We label the set of symmetric module families as Γ, and we index individual modules of such a family $M \in \Gamma$ as $M^1, M^2, \ldots, M^{|\Gamma|}$. We refer to a family M^a containing only one module $|M^a| = 1$ as an *asymmetric module*.

A family of modules $M \in \Gamma$ consists of multiple symmetric copies of the same module. We denote the number of modules in a family as $\#M = |M|$, and we can refer to the specific copies as $M_1, M_2, \ldots, M_{\#M}$. We can omit the index if it is clear to which module we are referring or if we are interested in *any* module of this type.

The local variables of a family of modules M are denoted by $var(M)$ while we use *global* to denote the set of global variables. We use indexing to refer to corresponding local variables, i.e. v_i is the copy of $v \in var(M_i)$. The n-th variable of a module is denoted $var_n(M)$, and similarly its copy in M_i is denoted by $var_n(M_i)$.

Commands are used to describe the behaviour of each member of a family of modules M. These are stated in terms of the local variables of module M_1, and the behaviour of any of the copies M_i (for $2 \leq i \leq \#M$) is derived by substituting each variable v_1 with the corresponding variable v_i. Each command has an optional synchronisation label a, which can be used to show that it belongs to a family of synchronised commands. This label is not present in commands that are not synchronised and translation rules for those commands will omit this label. Statements are guarded commands: they consist of a *guard* which determines whether the statement is executable in a given state, and a *stochastic update* that details the effect of executing the statement. Guards are boolean expressions over the global variables or the variables local to that module, while updates take the form $e^1 : u^1 + e^2 : u^2 + \cdots + e^l : u^l$ where the e^i are expressions that evaluate to a numerical value giving a probability and u^i is the update associated with that probability. Each update u^i can either do nothing (`true`) or be a set of simultaneous changes to distinct variables, either local to M or global. If a synchronisation label is set, then the updates can only affect local variables. The expressions listed above, $\mathsf{expr}(M_i)$, are subdivided into two types: $\mathsf{local\text{-}expr}(M_i)$, which allows only the use of local variables, and $\mathsf{symm\text{-}expr}$, which is a *fully symmetric expression*. The latter allow the use of constant values, global variables, or a conjunction of a local expression over *all* copies of a module.

Additionally, we use $S(M)$ to signify the local state space of a module M. This state space is the same for any copy of M and we let t be such that $t = |S(M)|$. As the set of states for a module M is countable, there exists a bijection $f_M : S(M) \to \{1, 2, \ldots, t\}$. The subset of local states of M which satisfy an expression e are denoted by $\mathsf{SAT}_M(e) = \{l \in S(M) : l \models e\}$.

Chapter 4

Preliminaries 2: Ctrl-MAC

Our main initial goal was to use formal methods to analyse the Ctrl-MAC communications protocol. This protocol was designed as a novel network protocol for WA-CPSs that is the first to support the control of multiple simultaneous physical processes requiring bidirectional communication [22]. It was developed by our colleagues at Imperial College London as a part of the S4 project (see Section 2.2.2). In this chapter we describe the protocol in detail.

Ctrl-MAC was developed as part of the design of a slow-loop controller and communication system for large area infrastructure. As previously mentioned, this type of system is called a Wide Area Cyber-Physical System, and examples include smart urban water distribution systems, precision agriculture and the electrical power grid.

A controller can keep a system stable if it can receive sensor information and send control updates in a timely and reliable fashion. Previously, these control systems would depend upon wired sensors and actuators. The wired approach cannot scale well to large area infrastructure due to the costs imposed by the length of wires needed [22]. The solution was to switch to wireless communication between the sensors and actuators, which would additionally reduce the deployment and maintenance costs. However, existing wireless technologies did not support the communication requirements in terms of communication range and transmission delays. Existing wireless control protocols, e.g., WirelessHart, support maximal ranges of up to 100 meters, which is insufficient for large area infrastructure [218]. While multi-hop protocols, e.g., 6TiSCH, could be used to extend the communication ranges sufficiently, this would be achieved at the cost of communication reliability [228]. However, communication reliability is tightly coupled with communication delays, and multi-hop systems increased these delays to a level that was unacceptable in terms of controller stability. For this reason single-hop networks are preferred for control systems [162, 228]. LPWA technologies [202] were chosen as the basis upon which to build the solution for this WA-CPS challenge. Previous LPWA technologies (e.g., LoRa, NB-IoT) have been designed with respect to monitoring systems; supporting one directional communication from sensors to gateway. WA-CPS control applications require support for two directional communication: i.e. additional communication from the gateway to the ac-

tuators. This would result in problematic large non-deterministic communication delays [162], so Ctrl-MAC was designed with the necessary adjustments to resolve this issue.

The main goal of Ctrl-MAC is reliable communication in a timely fashion. To achieve this, it was crucial that Ctrl-MAC is designed with system constraints and control and communication parameters that are specific to WA-CPSs. Both the communication system and the control system need to be considered during the design due to their influence on each other.

Single-hop LPWA networks which are not designed for control purposes lack in terms of delay bounds, message loss, two-way traffic and duty cycling [22]. To guarantee the stability of the control system, its event response time must meet an upper bound imposed by the system use case. This service time is directly related to the message delays and message losses of the system. Duty cycling is a restriction on the maximum percentage of time that a node can transmit on a channel. Wireless communication typically uses a shared part of the wireless spectrum that is governed by fair usage rules [22]. For example, a 1% duty cycle would require devices that broadcast for 1 second, to be unable to broadcast again for another 99 seconds. Ctrl-MAC and the related control model address these issues and demonstrate a workable resource constrained solution for control in WA-CPSs.

4.1 Technical description of Ctrl-MAC

The operation of Ctrl-MAC is divided into two phases: the *sensing and data transmission phase* and the *control update and actuation phase*. The sensing and data transmission phase handles the process of gathering environment data by the sensors and transferring the data to a central gateway node. The gateway node then calculates the appropriate control actions based on the information received from the sensor nodes, and instructs the actuator nodes accordingly in the control update and actuation phase. The main reliability requirement is that the sensors are able to send data in a timely manner, so the formal analysis will focus on the first phase.

Wireless communication of sensor devices makes use of channels of different frequencies when transmitting data. These channels have a duty cycle imposed on them, i.e. a restriction on the maximum percentage of time that a node can transmit on a channel, and there is also an EU recommendation on their total number [48]. Different channels can have different duty cycles. Ctrl-MAC assumes that there are three 1% duty cycle channels and one 10% duty cycle channel (see Fig. 4.1). The 10% channel is used for data transmission requests and actuator updates for the actuator nodes, while the other channels are used for data transmission.

The sensing and data transmission phase is governed by periodic Request-Reply Messages (RRMs) (see Figure 4.1) sent out by the gateway device. These RRMs partition time into intervals which we call RRCs. Each RRC begins with an RRM and lasts for a predefined amount of time. In this phase, sensor nodes progress through three stages.

First, each sensor node senses the physical environment to identify whether an event has

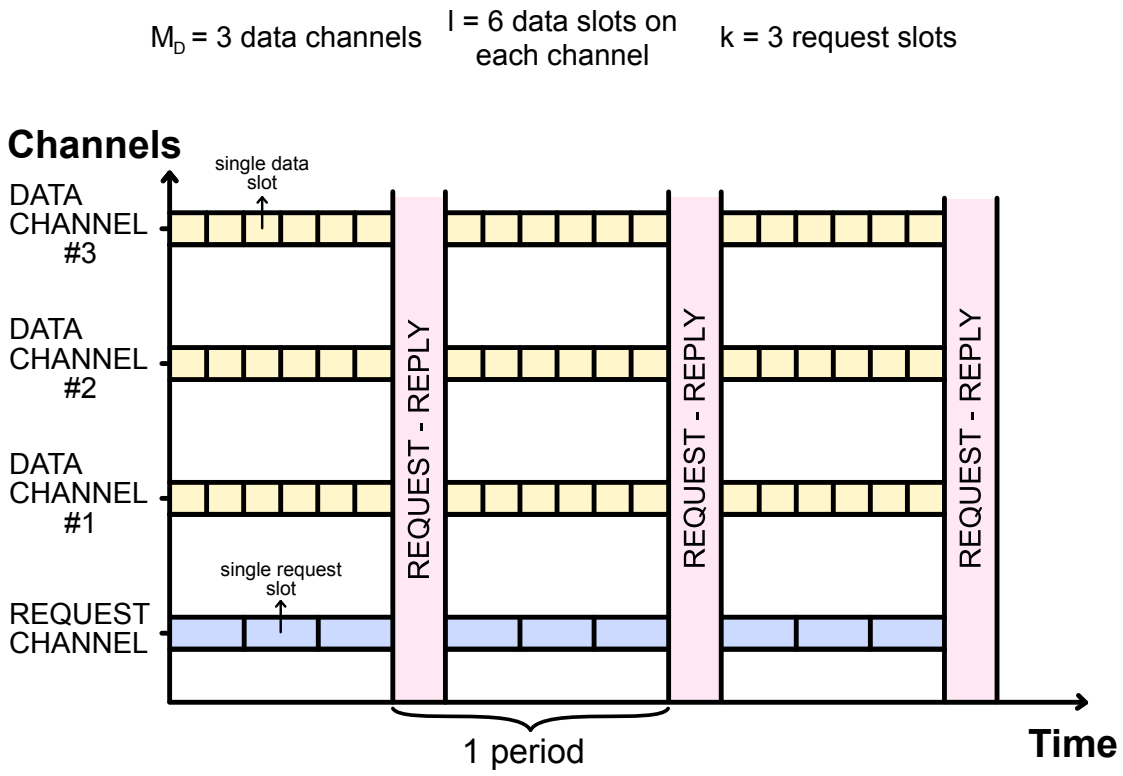


Figure 4.1: One request channel and three data channels, with RRM segmenting time in periodic intervals.

occurred. Events are defined as a pattern in the sensed data; e.g., the data going above or below a predefined threshold, or experiencing a change above a predefined magnitude. If an event is detected, the sensed data needs to be transmitted, otherwise the sensor sleeps until the next sensing cycle. If data needs to be transmitted, the sensor node first waits for the next RRM and uses it to synchronise with the gateway by updating its local device clock to the start of the RRM. This is necessary, as clock drift may occur when sensor nodes remain in a deep sleep power-saving mode over multiple sensing periods. At this point the sensor learns the number of request slots k, and the duration of each request slot t_{slot}.

In the second stage sensors transmit their data transmission requests. A sensor node chooses one of the k request slots at random and transmits a data transmission request in that slot. The choice is made using a uniform random distribution of all available request slots rather than a fixed choice or via round-robin scheduling in order to support cases involving more sensor nodes than request slots. A data transmission request constitutes of a sensor node transmitting its ID during the time interval of the chosen request slot. A request is successful if exactly one node transmits during a request slot, otherwise there is a collision (i.e. congestion has occurred in the associated request slot) and the requests of all sensor nodes which have chosen that request slot will fail.

Sensor nodes learn about the result of their data transmission request in the next RRM. Fig.

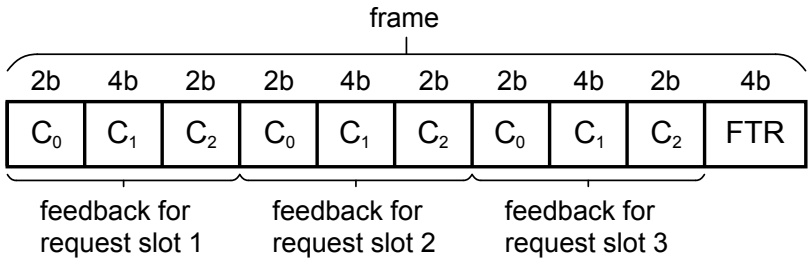


Figure 4.2: The structure of an RRM with $k = 3$ transmission request slots

4.2 shows an example of an RRM for an instance of Ctrl-MAC with $k = 3$ request slots. The frame of an RRM consists of three fields for each transmission request slot and one field for the FTR counter. The three fields for each request slot are used to provide feedback for that request slot and are interpreted as follows:

1. The first field, $C_0 \in \{0,1,2\}$ labels the state of the request slot. Request slots that were not used by any sensor nodes are indicated by $C_0 = 0$; $C_0 = 1$ indicates that a single sensor node has made a request in the request slot, and thus that request was successful; while $C_0 = 2$ signifies that a collision has occurred in the request slot.

2. The second field is the data slot counter, $C_1 \in \{1,...,l\}$. Here l is the total number of data slots. This field tells sensor nodes with successful requests which data slot they should use to send their data.

3. The third field is the data channel slot counter, $C_2 \in \{1,...,M_D\}$. It is used to inform successful sensors of the channel of their allocated data slot. Here M_D is the number of data channels. Apart from the data transmission request channel and the actuator downlink channel, all channels are data channels, i.e. $M_D = M - 2$, where M is the total number of channels used by Ctrl-MAC.

The FTR counter is the cumulative sum of contentions that have not been resolved at the time of the request reply message. This value is incremented for each congested request slot and decremented after each RRM. Sensor nodes interpret an RRM using the process described below.

The value of C_0 is checked for the request slot used by the sensor node. If $C_0 = 1$, the sensor node proceeds to stage three and sends its data. If $C_0 = 2$, the sensor node has to perform another data transmission request. Before it does so, sufficient back-off time needs to be applied in order to prevent network congestion. The time for the next data transmission request is determined based on the value of the FTR. The sensor node counts the number of all other request slots with $C_0 = 2$. This value is denoted by r, indicating that there are $r + 1$ total slots that have experienced contention. Each sensor node determines the number of congested request slots before its own in the RRM. This value is denoted as p with $p \in \{0,...,r\}$. The sensor then waits for $(FTR + r - p)$ RRMs to pass from the last data transmission request before another request

is made. For example, assume that $FTR = 0$, and the current RRC has had two request slots that have experienced a collision. The gateway generates and transmits the corresponding RRM. All sensor nodes that had chosen one of those two request slots count $r = 1$ other slots with contention. The sensors having chosen the first request slot find that $p = 0$ of those are placed in front of their own request slot, while those that have chosen the other calculate that $p = 1$. The sensor nodes of each group must then wait for

$$(FTR + r - p) = 0 + 1 - 0 = 1$$

RRM and

$$(FTR + r - p) = 0 + 1 - 1 = 0$$

RRMs respectively. This means that the sensors in the second group would transmit a new data transmission request in the current RRC, while those from the first group would do so in the following one. All sensors must transmit a request during their corresponding RRC. Each sensor individually chooses which request slot to send the request in, at random and with equal probability. Note that after the transmission of this RRM, the new value of FTR is 1. The gateway first increases the FTR value by 2 (as two collisions have occurred) and then decreases it by 1 (as an RRM has occurred). A good way to think about the FTR is that it is the number of sets of backed off sensors with respect to the number of RRMs they are backed off for. The formulas for the newly assigned back-offs assign all sensors that have made a transmission request in a particular congested request slot to a unique set, i.e. if those sensors are backed off for, say, 4 RRMs, then they will be the only sensors that will be backed off for 4 RRMs, and while their back-offs expire, no newly backed off sensors will be assigned their current back-off duration. As a corollary, the FTR value is equal to the largest currently assigned back-off duration.

Stage three is when the sensor node transmits its data. The RRM partitions each of the M_D data channels into l data slots. Each sensor node that has had a successful data transmission request is assigned a unique slot/channel pair by the gateway, which ensures that no contention will occur during data transmission. A sensor node which has received an RRM with $C_0 = 1$, changes its radio to the frequency of channel C_2 and transmits its data during time slot C_1. Additionally, if new data that triggers the event condition of the sensor has been sensed since initiating this process, only the latest data is sent to the gateway. The controller only needs the newest information to ensure the stability of the control system.

The control update and actuation phase of Ctrl-MAC is relatively simple and consists of periodic downlink messages sent by the gateway that deliver the necessary actuation information. As we previously noted, this phase will not be the focus of formal verification in this thesis, we refer instead to a more detailed description in [22].

Our focus is in modelling stage two of the sensing and data transmission phase. Stage one

requires all sensor nodes to wait for exactly one RRM before making their first data transmission request which does not have a meaningful impact on the performance of the protocol. The rate at which events occur has a significant impact on the protocol's theoretical performance and is discussed in Section 5.3.2. For the majority of this thesis we will consider the two extreme cases: either no events after the initial ones the model starts with, or a new event occurs immediately after a sensor device successfully transmits its data. In stage three the sensor nodes transmit their data in their allocated channel and data slot. The gateway assigns each sensor that has performed a successful data transmission request with a unique channel/slot pair. Therefore, no congestion should occur on any of the data channels, and we can assume that once a sensor receives data channel and data slot information in an RRM, it can successfully transmit its data during the RRC following that RRM.

Chapter 5

Initial Ctrl-MAC verification

As we previously stated, our main initial goal was to use formal methods to analyse the Ctrl-MAC communications protocol. We introduced the protocol and gave a technical description of its structure in Chapter 4. In this chapter we present our initial models and observations. We introduce a suite of models that we have incrementally developed in applying formal verification to Ctrl-MAC, and we present an approach that could allow us to scale our verification to the required number of sensors.

5.1 Initial Ctrl-MAC PRISM models

Ctrl-MAC was first presented in [22] where its correctness was demonstrated via simulation and statistical analysis. While this approach was able to show that Ctrl-MAC operated correctly for a finite set of runs, it could not provide a guarantee of performance in all scenarios. To complement the simulation results we will use PRISM [143] to analyse the protocol. Formal methods have been shown to be a useful tool in protocol analysis (e.g. the Chord routing protocol, which was designed without the use of formal methods, and the issues subsequently discovered [161, 247] and see Section 2.3 for other examples).

In the simulation and statistical analysis of Ctrl-MAC [22] the total delay introduced by Ctrl-MAC t_{MAC} is defined as the sum of the delays introduced by its stages, $\{t_{sync}, t_{req}, t_{send}, t_{update}\}$. Out of these, the delay of the synchronisation process t_{sync} is a fixed delay based on the size of the RRC, while the delays of the data transmission stage t_{send} and the actuator control input phase t_{update} are deterministic due to the use of dedicated data slots and channels. We assume that there is no network traffic or interference during those three stages and phases as due to the nature of TDMA protocols it can be expected that only the gateway device broadcasts during the time allocated for the RRM, and that there is no contention of the network during allocated channel and time slots. We focus our modelling efforts on the non-deterministic component t_{req}, the delay associated with Stage Two. This delay has the biggest impact on t_{MAC} as it is not bounded and it has previously been only analysed through numeric simulation. We attempt

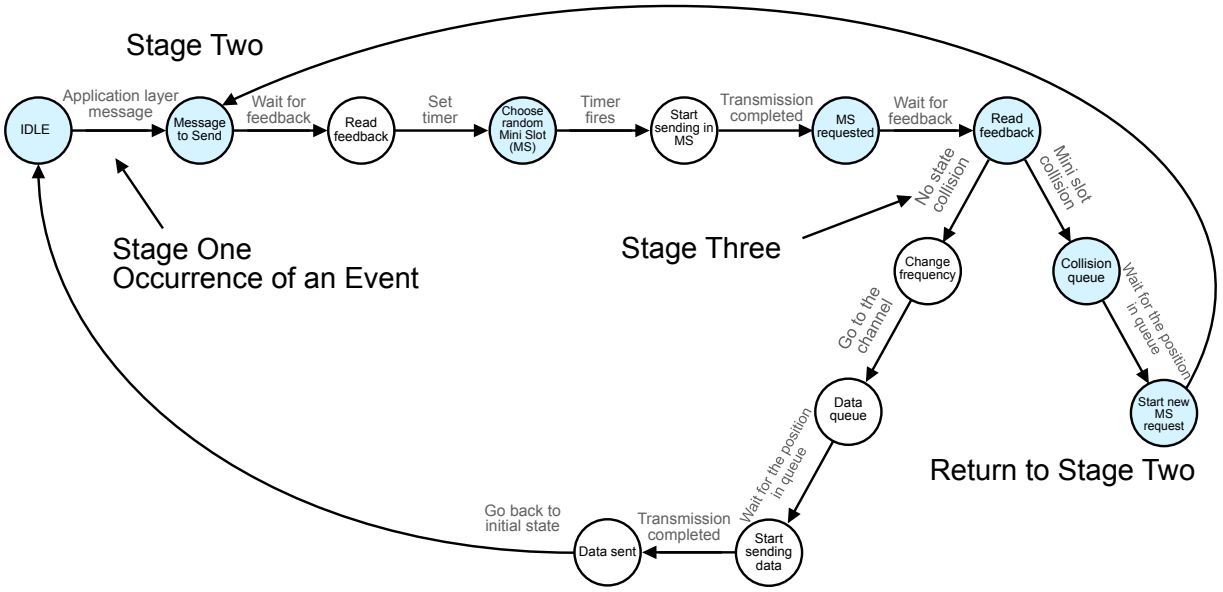


Figure 5.1: States of a Ctrl-MAC sensor device. States used in our PRISM models are coloured in blue.

to model the network traffic and interference present at the core of this stage in order to obtain some formal proofs about its time bound. We assume there is no noise present in the network as we expect the effect of the noise to be minimal in comparison to that of network traffic.

We have chosen to model Stage Two as a DTMC. The periodic nature of the protocol ensures that concurrency can be omitted, i.e. the time necessary for a device to perform a particular action is sufficiently smaller than the time interval assigned for that action. Furthermore, if congestion occurs, the order in which the messages occurred does not matter, only the slot they belong to. We use the time division of the protocol as a basis for the discretisation of the model.

For our initial PRISM model we applied a naive and straightforward approach to modelling the protocol. We attempted to represent each of the devices participating in the Ctrl-MAC protocol as separate entities. Ctrl-MAC's operating principle divides devices into two categories: a gateway device that is responsible for collecting all of the data to be used by the actuators of the CPS, and sensor devices supplying that data. Each of these sensor devices follows the same workflow and the operation of such a device is depicted in Figure 5.1. Note that for our models we do not include the states associated with Stage Three of the Ctrl-MAC protocol. We assume that once a device has a data slot assigned to it, there will be no congestion on that data channel in that time slot. In practice, issues in transmission can always occur but we will focus on the time it takes for a participating device to receive a time slot. The gateway device contains the logic necessary to allocate data transmission time slots to the sensor devices.

Our initial PRISM model was inspired by an existing PRISM model used in the verification of IEEE 802.11 Wireless Local Area Network Protocol [146]. The modules used for our PRISM model corresponded to the function of each device, so they also could be divided into two types:

a gateway device, and a sensor device. There is only one gateway device, so the model has only one gateway module. This module models the RRMs, i.e. the status of each request slot and the value for the FTR. The type used to represent the sensor devices only models the request status of a specific device and follows the states depicted in Figure 5.1. As all sensor devices follow the same procedure, PRISM module renaming was used to create multiple copies of the same module for the desired number of sensor devices.

The module used to represent the gateway needs additional logic to generate RRMs and to determine which modules have had successful requests and which have experienced contention. Guarded commands were used to update a device's state, with the guards synchronising the updates in the way they would be done by the RRMs. Our initial PRISM model can be found in Listing 5.1. The number of requests received during each request slot is stored in a global variable `c0`, `c1`, `c2`, etc. An RRM is generated by updating the local variables `C00`, `C10`, `C20`, etc to represent the statuses reported by an RRM (see Figure 4.2) and by calculating the corresponding value of the FTR. The global variables are reset, so that they can be used to count requests received during the new RRC. In this way, each sensor device both knows the information necessary to assign its back-off duration, and is able to transmit new requests during the RRC.

```
1  module gateway
2
3  active1: [0..1] init 1; //if 1 then ready to send request_reply message
4  active2: [0..1] init 1; //if 1 then not yet decremented FTR
5  x : [0..TIME_MAX]; //clock for gateway
6  //local state
7  g: [1..3] init 1;
8  //1 at start of request-reply slot
9  //2 receiving data and request messages
10 //3  generate schedule
11 C00 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
12 C10 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
13 C20 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
14 C30 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
15 C40 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
16 FTR : [0..MAX_FTR] init 0;
17
18 [request_reply] g=1 & x = 0 & active1=1 & active2=1-> (active1'=0); //
      synchronise with the nodes, relay schedule
19 [] g=1 & x=0 & active1=0 & active2=1 -> (FTR'=newFTR) & (active2'=0); //
      decrement FTR if appropriate
20 [time] g=1 & active1=0 & active2=0 & x<REQUESTSLOT_TIME-1-> (x'=x+1); //
      increment time through RR slot
21 [time] g=1 & x=REQUESTSLOT_TIME-1-> (x'=x+1) & (g'=2); //increment time out
      of RR slot
22 [time] g=2 & (x<TIME_MAX-1) -> (x'=x+1); // move through remaining slots,
```

```
   receive data
[time] g=2 & (x=TIME_MAX-1) -> (g'=3); // approach next RR slot
[] g=3 & x>0 -> (x'=0) & (FTR'= min(FTR + new_contentions,MAX_FTR));
[] g=3 & x=0 -> (C00'=c0) & (C10'=c1) & (C20'=c2) & (C30'=c3) & (C40'=c4) &
          (active1'=1) & (active2'=1) & (c0'=0) & (c1'=0) & (c2'=0) & (c3'
   =0) & (c4'=0) & (g'=1); //generate schedule
endmodule
```

Listing 5.1: PRISM code for gateway module in model 1.

Our initial PRISM model (which we henceforth refer to as model 1) was not optimised. Listing 5.2 shows an improved version of the same module (which was used in our next model, model 2). The variable which was used to keep track of time was completely removed. All request slots in a request-reply cycle have the same duration, so it would be possible to track (a discretised form of) time by only monitoring the current request slot. Furthermore, while the request slots (and the sensor nodes' requests for each slot) happen sequentially, our PRISM model can model them occurring simultaneously as the choices of request slots are independent of each other (each sensor node performs a random choice). The same logic was applied to the sensor modules, so that sensors can communicate their choice of request slot at any point during the RRC as opposed to only during the time interval associated with that request slot. Additionally, whereas in model 1 the gateway node kept track of its local state through the use of three local variables: `active1`, `active2`, and `g`, this was simplified to a single variable `g` in model 2. Full versions of both models can be found in Appendix A.2 and Appendix A.3.

```
module gateway
//1 at start of request-reply slot
//2 receiving data and request messages
//3  generate schedule
g: [1..4] init 1;

C00 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C10 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C20 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C30 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C40 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
FTR : [0..MAX_FTR] init 0;

[request_reply] g=1 -> (g' =2); // synchronise with the nodes,relay schedule
[] g=2 -> (g'=3);
[time] g=3 -> (g'=2); // sensors have chosen request slots, can generate RRM
[] g=4 -> (C00'=c0) & (C10'=c1) & (C20'=c2) & (C30'=c3) & (C40'=c4) &
          (c0'=0) & (c1'=0) & (c2'=0) & (c3'=0) & (c4'=0) &    // generate
          (FTR'=min(newFTR+new_contentions,MAX_FTR)) & (g'=1); // schedule
endmodule
```

Listing 5.2: PRISM code for gateway module in model 2.

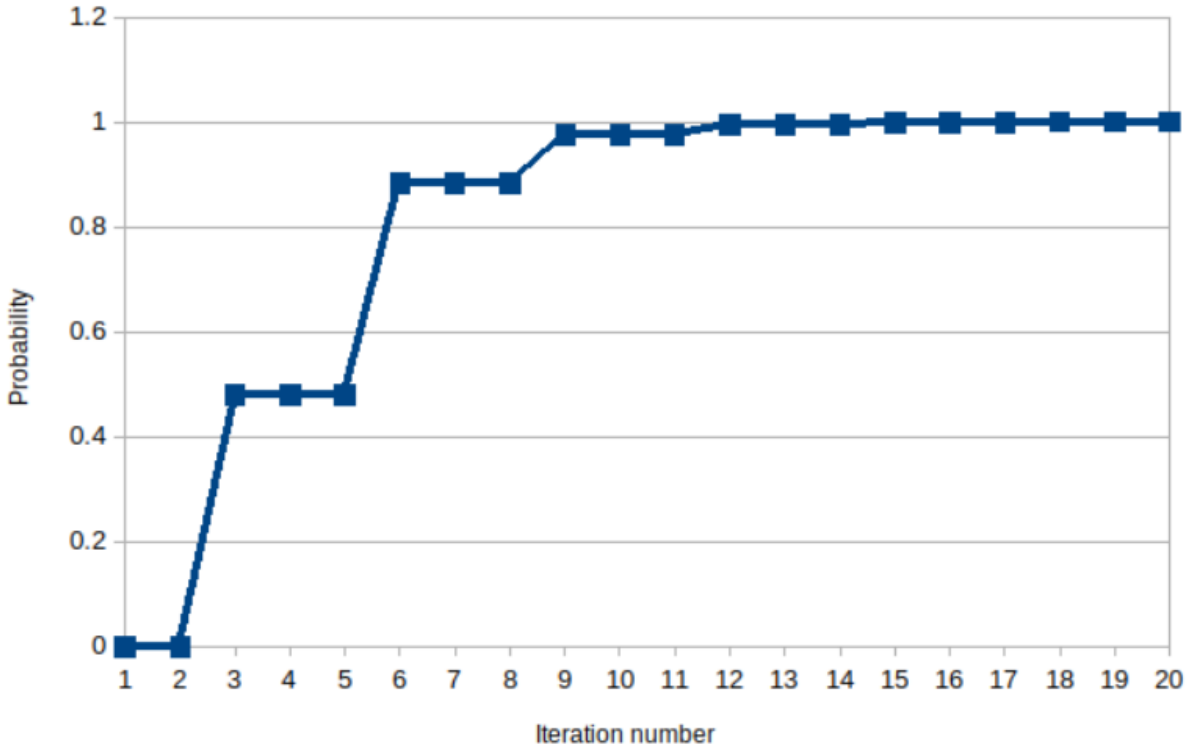


Figure 5.2: A graph of the results obtained from model 2. Probability of successful data delivery of a specific sensor is plotted against number of RRMs elapsed. Results are for a model with 3 sensor devices, which all start in a state where an event has been sensed, and no new events are registered.

Both model 1 and model 2 allow us to specify properties for each individual device, such as verifying that “the probability that a sensor device is able to eventually send its data is 1”, calculating "the probability that sensor device A is able to send data within ten RRCs occurring", and computing "the average number of RRCs that occur between data generation and data transmission". Furthermore, PRISM’s simulator can be used to investigate paths that lead to a state of interest, i.e. one that violates or satisfies a given property. In Figure 5.2 we present some example results produced by this model. The scenario modelled is for an instance of Ctrl-MAC with three participating sensor devices, where all three devices are attempting to send data. No other devices detect data surpassing their threshold for the purposes of this model. The graph shows the probability that one of the sensors delivers its data successfully after the specified number of RRMs have occurred. Table 5.1 shows model computation times and sizes for a range of sensor devices involved in a Ctrl-MAC protocol for model 2. All experiments presented in this chapter were performed on a 2.60 GHz PC with 16 GB RAM, running PRISM version 4.8 under Windows. The maximum memory of the CUDD library was set to 1 GB (PRISM default) and the Java maximum memory was set to 6 GB. Build times, number of states and transitions increase by an order of magnitude when a sensor is added. The model runs out of available memory when run for an instance with 8 participating sensor devices.

Number of sensors	Time (s)	States	Transitions
1	0.061	52	66
2	0.169	343	687
3	0.331	4,097	11,183
4	0.626	45,362	164,106
5	1.655	518,105	2.35e+6
6	11.36	2.12e+7	8.57e+7
7	68.76	3.59e+8	1.65e+9
8	mem-error	mem-error	mem-error

Table 5.1: States, transitions and build times for PRISM models of Ctrl-MAC with the specified number of sensor devices based on model 2.

Developing these two models resulted in a number of beneficial outcomes. At the most basic level, it aided the communication with the protocol developers, allowing us to find and visualise many cases of interest. In particular, we were able to discover a number of inconsistencies and ambiguities in the initial protocol specification the developers provided us with. For example, we were able to make clear whether counting the number of request slots started from 0 or from 1, which could result in an off-by-one error when assigning back-off periods. Additionally, we identified a number of mistakes in some of the formulas in the original protocol specification, and cleared up an ambiguity concerning whether the FTR counter gets decremented before or after an RRM is sent out. There was also an undocumented case for what actions sensor devices, that have experienced unsuccessful transmission requests, should take if FTR equals 1. When Ctrl-MAC is described in [22] it is said that FTR "includes the current failed and previous failed transmission requests". Our models showed that this would lead to double counting of the current failed transmission requests in calculating the back-off of a sensor node using the $(FTR + r - p)$ formula. The `FTR` in this formula actually only refers to previous failed transmission requests as the current failed ones are counted by each sensor node via the variable `r`. The value of FTR is updated to include the current failed transmission requests only after the back-offs have been calculated. This particular finding was essential for the improvements we made in our next model, namely model 3 (see below). While these examples might look trivial to people well-acquainted with Ctrl-MAC, they can cause a good deal of confusion for software developers new to the protocol. Developing a PRISM model alongside the Ctrl-MAC specification was of great benefit to both us (the modellers), and the developers. Even at this stage the formal analysis allowed both sides to gain a much deeper insight into the communication protocol and discover a range of issues with the protocol specification.

The structure of the two models followed the operating principle of Ctrl-MAC, outlined in Section 4.1. During the RRM portion of each RRC the sensor nodes do nothing while the gateway node calculates the values used in the RRM. Conversely, for the rest of the RRC, the gateway only listens, while the sensor nodes choose their slots and transmit their requests. The gateway and sensor node modules model this behaviour by alternating which one performs ac-

tions. The two module types are synchronised by two labels, `request_reply` and `time`, which serve to distinguish the transitions of the gateway module from those of the sensor modules, so that all sensor nodes have finished making their transmission requests at the point that an RRM is generated. Similarly, sensor nodes do not choose request slots and make requests while the gateway is generating an RRM. Using this observation it is possible to greatly simplify the modules by restructuring their order of operation. Previously, the order was (starting at the beginning of an RRM):

1. Gateway module determines which request slots had experienced contention (i.e. local variables `C00`, `C10`, etc) based on how many sensor modules had transmitted a request during those request slots (i.e. global variables `c0`, `c1`, etc) and updates the FTR accordingly.

2. Gateway module resets the global variables `c0`, `c1`, etc to keep count of requests received during the next RRC.

3. Sensor modules check if their request was successful, and calculate the corresponding back-off if it was not. This back-off duration is based on the number of congested request slots from the previous RRC (i.e. variables `C00`, `C10`, etc).

4. Sensor modules that are not currently backed off choose a request slot at random and generate a transmission request by updating the corresponding counter (i.e. on of the global variables `c0`, `c1`, etc).

This ordering meant that information about the request slots for both the previous and current RRC had to be maintained. This is because one was needed to assign back-offs while the other to perform transmission requests. This doubled the variables required to model the request slots. Consider instead the following ordering of events (again starting at the beginning of an RRM):

1. Sensor modules check if their request was successful by looking at the number of requests transmitted in the request slot chosen by them (i.e. global variables `c0`, `c1`, etc). If unsuccessful the corresponding back-off is calculated based on those variables and the current FTR using the $(FTR + r - p)$ formula.

2. Gateway module updates FTR based on the number of slots that have experienced contention (which is determined by the number of requests received during each request slot, i.e. `c0`, `c1`, etc).

3. Gateway module resets the global variables `c0`, `c1`, etc to keep count of requests received during the next RRC.

4. Sensor modules that are not currently backed off choose a request slot at random and generate a transmission request by updating the corresponding counter (i.e. on of the global variables `c0`, `c1`, etc).

This alternative ordering removes the requirement to keep track of the state of request slots in the previous RRC. In turn, this allows us to remove the local variables `C00`, `C10`, etc. from our model, which greatly simplifies the gateway module as shown in Listing 5.3. Table 5.2 shows the improved model construction times and the reduced model sizes resulting from this improvement (which we henceforth refer to as model 3; full version available in Appendix A.4). The number of states and transitions increase by an order of magnitude with the addition of a new sensor device; however, the build time now increases by an order of magnitude every two additional sensors. We note that model 3 was able to be constructed for up to nine (two additional) sensor nodes before the memory limit was reached.

```
module gateway
//1 at start of request-reply slot
//2 generate schedule
//3 receiving data and request messages
g: [1..3] init 1;

FTR : [0..MAX_FTR] init 0;

[request_reply] g=1 -> (g' =2); // synchronise with the nodes,relay schedule
[] g=2 -> (c0'=0) & (c1'=0) & (c2'=0) & (c3'=0) & (c4'=0) &    // generate
          (FTR'=newFTR) & (g'=3);                              // schedule
[time] g=3 -> (g'=1); // RRM has been generated and broadcast
endmodule
```

Listing 5.3: PRISM code for gateway module in model 3.

Number of sensors	Time (s)	States	Transitions
1	0.037	34	48
2	0.010	264	608
3	0.157	3,498	10,584
4	0.363	43,467	164,275
5	1.12	537,940	2.44e+6
6	4.79	6.71e+6	3.56e+7
7	16.0	8.47e+7	5.12e+8
8	56.9	1.08e+9	7.31e+9
9	205	1.40e+10	1.04e+11
10	mem-error	mem-error	mem-error

Table 5.2: States, transitions and build times for PRISM models of Ctrl-MAC with the specified number of sensor devices based on model 3.

Clearly, given the naivety of our initial models, we could only derive results for a small number of sensors. This is a common problem though - all similar models on the PRISM website [193] are only applicable to a small number of processes. They are still extremely useful for analysing the behaviour of the underlying protocol. We do, however, want to investigate the

behaviour of Ctrl-MAC for larger systems, and the rest of this thesis is devoted to developing ways to do this. Meanwhile, the following observations explain why our state space explodes so rapidly.

One main reason for our large state space is that sensors are treated as distinguishable objects, so each sensor performs its choice for which request slot it uses in a transmission request individually. This results in the total number of states being a multiple of a factor that depends on the total number of request slots. During each RRC the PRISM model considers sensor device A choosing the first request slot and sensor device B choosing the second request slot as a different case from sensor device A choosing the second request slot and sensor device B choosing the first request slot. Furthermore, the different orders in which devices A and B make their choices lead to different paths in the model. We previously mentioned partial order reduction as a solution to models where interleavings of independent events are equivalent, and have seen it applied to symbolic model checking [5]; however, to our knowledge, this is not supported by PRISM. However, in Ctrl-MAC there is no operational difference between sensor devices: no data is given priority and no data delay is less desirable than another. The protocol considers only the number of sensors choosing each request slot, so all of the cases above result in the same overall behaviour. The behaviour of the model only depends on how many sensors have chosen each request slot. This in turn depends on the number of sensors that have transmitted transmission requests in that RRC. Our goal is to remove the unnecessary complexity introduced between these two steps.

The properties we want to verify for Ctrl-MAC do not reference all of the sensor nodes individually. We would be interested in more general network qualities: the probability that the number of collisions exceeds a given value, the expected number of recently failed transmission requests, the probability that *any* sensor is unable to send its data for longer than a specified threshold, etc. To do so we either want to consider all nodes generically (e.g. to verify that all sensors are able to send within a given time frame), or want to distinguish a single sensor but consider the remaining sensors generically (e.g. to find the probability that a sensor node is able to send its data by generating at most two transmission requests). Currently, our PRISM models refer to all of the sensor nodes individually. A WA-CPS is often not concerned with which exact device is being delayed or prevented from sending its data. It is generally much more beneficial to look at the overall system performance: i.e. the probability that any device is delayed by more than some number of cycles or the average delay between data generation and data transmission. We investigate symmetry reduction as a possible solution as it is an effective way of combating the state space explosion problem in similar cases and it can allow us to prove this type of generic properties about the overall system behaviour.

5.2 Ctrl-MAC PRISM model with manual counter abstraction

Although our initial models were useful, we clearly require a more scalable approach to allow us to generalise our results to a larger number of sensor devices. Exploiting the symmetry present in our models by merging all - or all but one - of the sensors has been identified as a potential way to reduce the size of the state space. We choose to apply symmetry reduction through the use of counter abstraction to achieve this. Using a counter to generically represent states of the original model has been shown to be effective in reducing the state space for symbolic model checking [71]. In this section we attempt to manually apply counter abstraction to the Ctrl-MAC model following the macroscopic approach used to verify swarms of foraging robots in [139]. In that scenario, individual behaviours defined at the microscopic level resulted in complex group behaviours at the macroscopic level. A counting abstraction was beneficial in modelling the overall group behaviour as the multiple individual processes were all identical. Similarly, in Ctrl-MAC sensor nodes are identical and obtaining information about group behaviour can be useful. For example, as the back-off assigned by sensors with unsuccessful transmission requests are closely related to the current FTR value, obtaining a value for the probability that FTR reaches some specified upper or lower bound could be beneficial in establishing a range for the back-off duration following an unsuccessful request. In Section 6.5.5 we will compare this manual approach to a more automatic approach using GRIP.

To do this for our models, we must describe counter variables for every state a sensor can be in, so that we keep track of how many sensors are in that given state. These counters include single states such as *idle*, *sending*, *sent*, as well as families of states describing sensors that have been backed off for a number of RRMs or sensors that have chosen to send on a particular request slot. The number of counters needed for the latter category is equal to the number of request slots, while the size of the former is bounded by the following observation: the maximum back-off is at most half the total number of sensors. When contention occurs, all sensor nodes that have chosen that request slot receive the same back-off duration, while sensors that are backed-off due to collisions in different request slots receive different back-off durations. As sensors that are backed-off cannot send new transmission requests, each sensor can only be backed-off in one group (in terms of their back-off duration). Therefore, as we need at least two sensors to choose the same request slot in order for a collision to occur, the maximum possible value for the FTR (and equivalently, the back-off) when there are n sensor nodes is

$$\left\lfloor \frac{n}{2} \right\rfloor - 1.$$

We note that the -1 term is because at each RRM the FTR is increased by the number of new contentions and simultaneously decremented as a result of the start of a new RRC. We also note

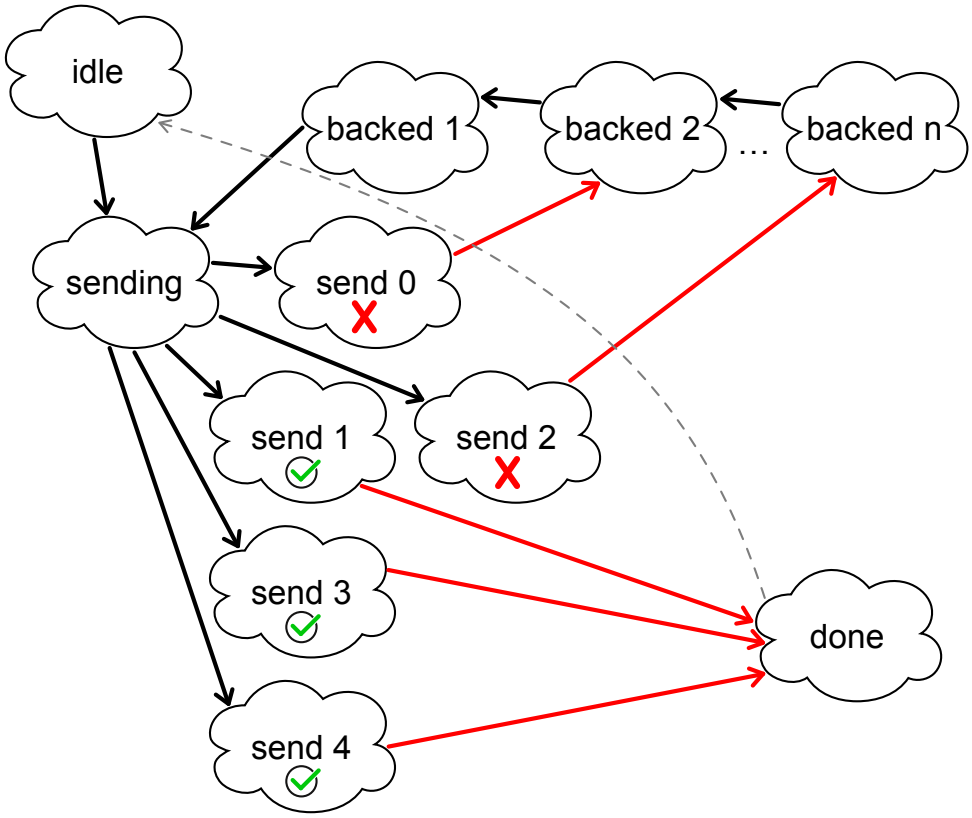


Figure 5.3: Counters and possible transition between counters. An arrow signifies sensors going from one sensor state to another. The dotted line represents the optional transition for models of non-bursty traffic.

that this extreme has a very low probability of being reached: in fact, for large numbers of sensor nodes this is also the case for a wide range of the largest FTR values. We discuss how this can be used to help the state space explosion problem in Section 7.1.

While the number of counters is less than the number of sensors, these values are still linearly related. The transitions between the states are simple to model: e.g. at each RRM the counter for sensors backed off for 1 RRM assumes the value of the counter for sensors backed off for 2 RRMs, etc. Figure 5.3 illustrates the counters used by the model and the ways in which those counters can change.

Listing 5.4 shows the gateway module with counter abstraction applied to it in this way. We have created new variables `idle`, `sending`, `send0`, `send1`, `send2`, `send3`, `send4`, `backed_off`, and `backed_off2` which keep track of the number of idle sensors, the number of sensors that have detected an event and will be transmitting a request in this RRM, the number of sensors that have chosen each of the possible request slots, and the number of sensors which are currently backed off for each possible back-off duration.

```
1  module gateway
2  g: [1..4] init 1;
3  FTR : [0..MAX_FTR] init 0;
4  // population counters
5  idle: [0..POPULATION] init POPULATION;// idle sensors
6  sending: [0..POPULATION] init 0;    // sensors sending requests this round
7  send0: [0..POPULATION] init 0;      // sensors having chosen request slot 0
8  send1: [0..POPULATION] init 0;      // sensors having chosen request slot 1
9  send2: [0..POPULATION] init 0;      // sensors having chosen request slot 2
10 send3: [0..POPULATION] init 0;      // sensors having chosen request slot 3
11 send4: [0..POPULATION] init 0;      // sensors having chosen request slot 4
12 backed_off1: [0..POPULATION] init 0;// sensors backed off for 1 RR cycle
13 backed_off2: [0..POPULATION] init 0;// sensors backed off for 2 RR cycles
14
15 [] g=1 & idle>0 & sending>=0-> (idle'=0) & (sending'=min(sending+idle,
      POPULATION)); // transition to attempt to send
16 // each sending sensor chooses the slot they like
17 [] g=1 & idle=0 & sending>0 ->
18 1/5:(sending'=sending-1)&(send0'=min(send0+1,POPULATION))&(c0'=min(c0+1,2))+
19 1/5:(sending'=sending-1)&(send1'=min(send1+1,POPULATION))&(c1'=min(c1+1,2))+
20 1/5:(sending'=sending-1)&(send2'=min(send2+1,POPULATION))&(c2'=min(c2+1,2))+
21 1/5:(sending'=sending-1)&(send3'=min(send3+1,POPULATION))&(c3'=min(c3+1,2))+
22 1/5:(sending'=sending-1)&(send4'=min(send4+1,POPULATION))&(c4'=min(c4+1,2));
23 // every sensor has chosen a slot
24 [time] g=1 & idle=0 & sending=0 -> (g'=2);
25 // change populations to specific backoff population based on congestion and
       current FTR
26 [] g=2 & c0>1 & send0>0 & (FTR+new_contentions-position0=1)->(send0'=0)&(
      backed_off1'=min(backed_off1+send0,POPULATION));
```

```
27 [] g=2 & c0>1 & send0>0 & (FTR+new_contentions-position0=2)->(send0'=0)&(
      backed_off2'=min(backed_off2+send0,POPULATION));
28
29 [] g=2 & c1>1 & send1>0 & (FTR+new_contentions-position1=1)->(send1'=0)&(
      backed_off1'=min(backed_off1+send1,POPULATION));
30 [] g=2 & c1>1 & send1>0 & (FTR+new_contentions-position1=2)->(send1'=0)&(
      backed_off2'=min(backed_off2+send1,POPULATION));
31
32 [] g=2 & c2>1 & send2>0 & (FTR+new_contentions-position2=1)->(send2'=0)&(
      backed_off1'=min(backed_off1+send2,POPULATION));
33 [] g=2 & c2>1 & send2>0 & (FTR+new_contentions-position2=2)->(send2'=0)&(
      backed_off2'=min(backed_off2+send2,POPULATION));
34
35 [] g=2 & c3>1 & send3>0 & (FTR+new_contentions-position3=1)->(send3'=0)&(
      backed_off1'=min(backed_off1+send3,POPULATION));
36 [] g=2 & c3>1 & send3>0 & (FTR+new_contentions-position3=2)->(send3'=0)&(
      backed_off2'=min(backed_off2+send3,POPULATION));
37
38 [] g=2 & c4>1 & send4>0 & (FTR+new_contentions-position4=1)->(send4'=0)&(
      backed_off1'=min(backed_off1+send4,POPULATION));
39 [] g=2 & c4>1 & send4>0 & (FTR+new_contentions-position4=2)->(send4'=0)&(
      backed_off2'=min(backed_off2+send4,POPULATION));
40 // change population to idle if transmission was successful
41 [] g=2 & c0=1 -> (idle'=min(idle + send0,POPULATION)) & (send0'=0);
42 [] g=2 & c1=1 -> (idle'=min(idle + send1,POPULATION)) & (send1'=0);
43 [] g=2 & c2=1 -> (idle'=min(idle + send2,POPULATION)) & (send2'=0);
44 [] g=2 & c3=1 -> (idle'=min(idle + send3,POPULATION)) & (send3'=0);
45 [] g=2 & c4=1 -> (idle'=min(idle + send4,POPULATION)) & (send4'=0);
46 // all requests have been sorted; make RR
47 [request_reply] g=2 & send0+send1+send2+send3+send4=0 -> (g'=3)
48   & (sending'=min(sending+backed_off1,POPULATION))
49   & (backed_off1'=backed_off2) & (backed_off2'=0);
50 // reset request slot counters
51 [] g=3 -> (c0'=0) & (c1'=0) & (c2'=0) & (c3'=0) & (c4'=0) &    // generate
52           (FTR'=newFTR) & (g'=4);                              // schedule
53 [rr_end] g=4 -> (g'=1);
54   endmodule
```

Listing 5.4: PRISM code for gateway module in model 4.

As mentioned previously, we are interested in verifying properties about the group behaviour established by the interactions between individual sensors. We separate these properties into two types. We refer to the first type as *generic* properties: e.g., sensors are able to eventually transmit data, or the probability that FTR obtains a value larger than 10 is less than 1%. These properties either reference an action performed by all of the identical processes or are related to a quantity that is not intrinsic to any of them. The second type of property are referred to as *quasi-generic*

Sensors	Time	States	Transitions
5	0.39	1660	3852
10	3.08	36,145	98,455
15	10.2	405,403	1.19e+6
19	22.0	2.20e+6	6.86e+6
20	25.6	3.24e+6	1.03e+7
25	52.0	1.86e+7	6.34e+7
30	108	8.19e+7	2.96e+8
35	237	2.95e+8	1.12e+9
40	error	error	error

Table 5.3: States, transitions and build times for the model with counter abstraction (model 4).

properties: e.g., the probability that a specific sensor waits for more than 50 RRCs from sensing its data to transmitting it is less than 2%, or the probability that any sensor is backed off more than four times in a row is less than 5%. This type of property distinguishes one of the identical processes but not the others. In fact this type of property is used to reflect a more general type of property, e.g. that *any* sensor has the required behaviour. However, as all of the sensors are identical, we use a specific sensor (e.g. sensor 1) to reflect the property of interest. This is required to express the property in PRISM.

A PRISM model with the gateway module from Listing 5.4 as its only module would be able to verify generic properties for Ctrl-MAC but would be unable to verify quasi-generic ones. Table 5.3 shows the state space and compilation times achieved by such a model (which we will refer to as model 4 henceforth; full PRISM file can be found in Appendix A.5). We note the drastic increase in the maximum number of sensors modelled before the memory limit is reached. Counter abstraction has enabled us to model the overall group behaviour of the protocol, but we have lost the ability to investigate how individual sensors progress through the protocol. While we do not want to identify all of the sensors individually in order to reduce the state space, it would be ideal to be able to identify one or some of them in order to use those for the verification of quasi-generic properties.

We use a hybrid approach to solve this issue. To model a Ctrl-MAC protocol with n sensor nodes, we create a PRISM model with two modules: a gateway module based on the one presented in Listing 5.4 which uses counter abstraction to represent the behaviour of $n-1$ sensor nodes, and one node module which individually represents the distinguished sensor node. Doing so enables us to achieve a portion of the possible state space reduction, while still allowing us to verify quasi-generic properties. Table 5.4 compares the performance (in terms of model build time) of this hybrid approach (which we will refer to as model 5) to the best performing PRISM model without symmetry reduction (model 3). We have tested both models on a suite of properties such as `P=? [F<50 s1=8]`, i.e. what is the probability that the individual sensor is able to transfer its data within 50 units of time, and in each case both models obtain the same result (but we have not included the result in the table). We observe that the addition of counter

Sensors	Time (1) (s)	States (1)	Transitions (1)	Time (2) (s)	States (2)	Transitions (2)
2	0.010	264	608	0.125	205	356
3	0.157	3,498	10,584	0.205	799	1635
4	0.363	43,467	164,275	0.457	2484	5653
5	1.12	537,940	2.44e+6	1.23	6753	16,523
6	4.79	6.71e+6	3.56e+7	2.44	15,977	41,388
7	16.0	8.47e+7	5.12e+8	3.95	33,332	90,214
8	56.9	1.08e+9	7.31e+9	7.4	76,256	218,712
9	205	1.40e+10	1.04e+11	15.5	141,281	416,365
10	mem-error	mem-error	mem-error	23.8	278,402	846,398
11	mem-error	mem-error	mem-error	33.1	498,837	1.55e+6
12	mem-error	mem-error	mem-error	42.9	849,112	2.69e+6
13	mem-error	mem-error	mem-error	56.8	1.40e+6	4.53e+6
14	mem-error	mem-error	mem-error	76.3	2.28e+6	7.46e+6
15	mem-error	mem-error	mem-error	116	3.61e+6	1.2e+7
16	mem-error	mem-error	mem-error	151	5.64e+6	1.90e+7
17	mem-error	mem-error	mem-error	273	8.67e+6	2.96e+7
18	mem-error	mem-error	mem-error	289	1.31e+7	4.54e+7
19	mem-error	mem-error	mem-error	356	1.96e+7	6.88e+7
20	mem-error	mem-error	mem-error	mem-error	mem-error	mem-error

Table 5.4: Comparison of model 3, the best performing model without counter abstraction (1), versus model 5, the hybrid model with counter abstraction (2).

Model Number	Short Description	Properties Checked
Model 1	Initial model	For each individual device
Model 2	Removed unnecessary details	For each individual device
Model 3	Optimised transition order	For each individual device
Model 4	Full counter abstraction	Generic properties
Model 5	Hybrid model	Quasi-generic properties

Table 5.5: Types of PRISM models that we have created.

abstraction doubles the number of sensor devices modelled before the memory limit is reached. Counter variables greatly reduce the rate at which the numbers of states and transitions increase as the number of sensor devices increases.

5.2.1 Performance comparison for different models

We will now present some of the verification results achieved using the five models described above (see Table 5.5 for a summary). These five models were run in two varieties based on the way the environment is modelled. The first considers bursty traffic and is based on models for the 2C protocol [166] where all sensor nodes simultaneously detect an event that meets their data transmission threshold and no further events are registered before all transmission requests are resolved. The one allows for new events to be detected by sensors that are not currently

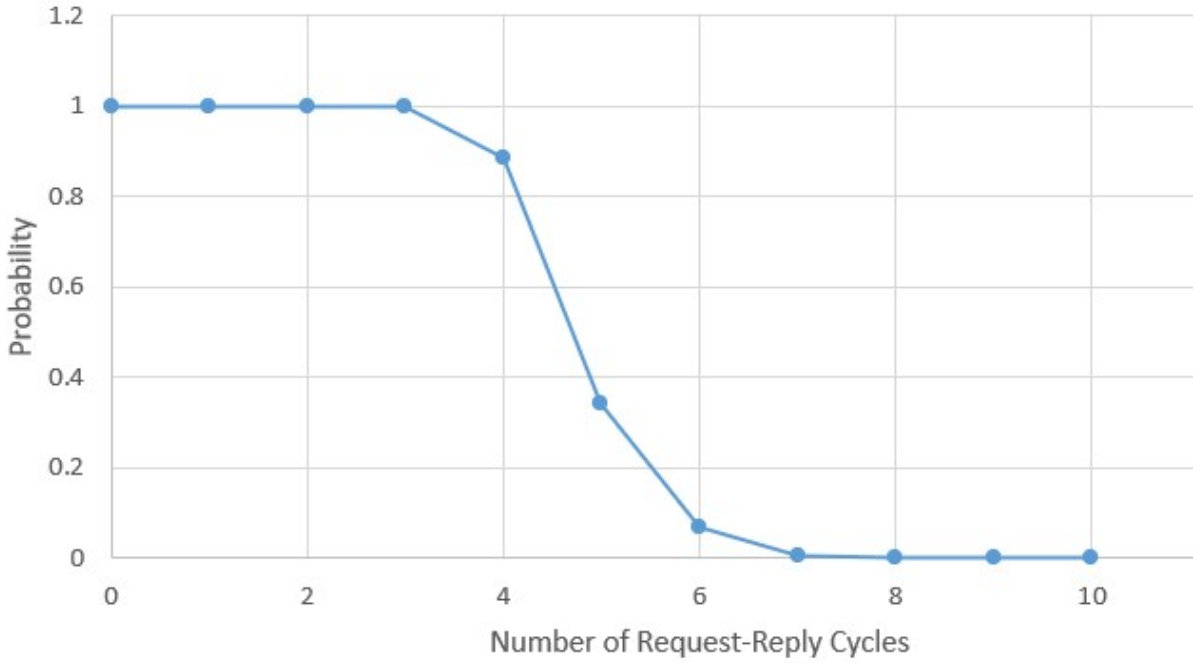


Figure 5.4: Verification of properties of the type `P=?[F(FTR=x)]` for the stated values of the number of RRCs, x. Performed using model 4, the fully counter abstracted model, on an instance with 19 sensor nodes.

attempting to transmit data. For these models we have assumed that the sensors are placed in an event rich environment and detect new events immediately after transmitting data from a previous event.

First, we present some of the generic properties we have verified. Properties such as the probability that the FTR reaches a specified value

$$P =? \ [F \ \ (FTR = x)],$$

where x is a value in the range of values of FTR, can be verified using model 4, our fully counter abstracted model, and thus can be checked for a larger number of participating sensors. Table 5.6 and Figure 5.4 show the verification results of this property for all values of x. We again note that in this model increments and decrements to the FTR occur simultaneously, so the theoretical maximum FTR value is

$$\left\lfloor \frac{19}{2} \right\rfloor - 1 = 8,$$

which is supported by the verification results. These properties show us that the top range of values assumed by the FTR are reached only with a low probability. The FTR is closely related to the assigned back-offs and these results can be used to infer the time between transmission requests, which in turn can be used to reason about the maximal number of failed requests possible before the transmission time limit is reached. We discuss this in more detail in Section

x	P=? [F (FTR=x)]	Time (s)
0	1.0	302
1	≈ 1	134
2	≈ 1	597
3	0.998	1840
4	0.884	21710
5	0.344	6086
6	0.070	7741
7	0.005	9081
8	1.004e-4	13512
9	0	0.047

Table 5.6: Verification of properties of the type P=? [F (FTR=x)] for the stated values of x. Performed using model 4, the fully counter abstracted model, with 19 sensor nodes.

5.3.1.

We can also model conflict resolution in the situation where messages are sent in an ad hoc fashion (rather than in bursts) by allowing sensors to transmit requests whenever they are not already involved in a transmission request cycle. To do this sensors that have completed a successful transmission request are returned to the idle state where they will begin a new transmission attempt (see Listing 5.5).

```
1 [] g=2 & c0=1 -> (idle'=min(idle + send0,POPULATION)) & (send0'=0);
2 [] g=2 & c1=1 -> (idle'=min(idle + send1,POPULATION)) & (send1'=0);
3 [] g=2 & c2=1 -> (idle'=min(idle + send2,POPULATION)) & (send2'=0);
4 [] g=2 & c3=1 -> (idle'=min(idle + send3,POPULATION)) & (send3'=0);
5 [] g=2 & c4=1 -> (idle'=min(idle + send4,POPULATION)) & (send4'=0);
```

Listing 5.5: Transition commands for sensors in the successful state to idle state.

Currently, idle sensor devices are set to instantly begin a new transmission request, which allows us to reason about the maximum throughput of the protocol, but the transitions from the idle to the sending state can be augmented with probabilities based on those observed for the specific Ctrl-MAC use scenario we are interested in. We achieve this by looking at steady-state properties, such as

$$S =? [FTR = x],$$

the long-run probability that the FTR has a value of x for some x. The verification results for this property for a range of values of x are depicted in Figure 5.5. In the equilibrium position, the FTR value is between 6 and 8 more than 90% of the time. As a consequence, 90% of the back-offs that are assigned to sensors experiencing congestion will be for between 6 and 8 RRMs. Equivalently, we find that the number of sensors that are not backed off is between 3 and 6, 90% of the time. This allows us to gain insight into the behaviour of the protocol: when supplied with a constant flow of data, the protocol issues back-offs to sensor nodes, such that the number of sensors that transmit during a single RRM is an optimal value that allows the

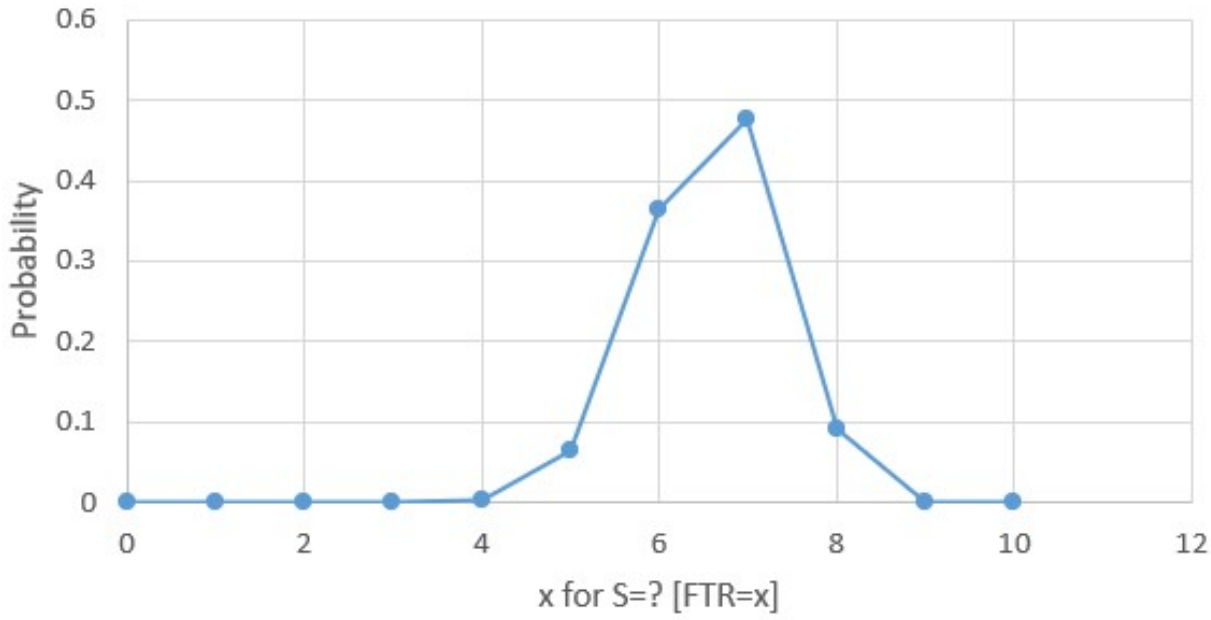


Figure 5.5: Verification of properties of the type `S=? [(FTR=x)]` for the stated values of x. Performed using model 4, the fully counter abstracted model, with 10 sensor nodes. Sensors instantly detect new events after they successfully transmit their data.

most successful transmission requests per RRM. This constitutes a negative feedback system, that uses a self-imposed limit to achieve the desired value in the long-run.

Verifying these properties for a range of models with a varying number of sensor nodes we find an emerging pattern. The negative feedback effect reduces the number of sensors sending transmission requests to a value between 3 and 5, which is a value that maximises the number of successful requests that occur in each RRC. We will calculate and discuss what this theoretical maximum value is in Section 5.3.3. At this equilibrium state, the number of transmission requests in an RRC is based on two events: the sensors which have been unsuccessful in their last attempt whose back-off counters reach zero, will necessarily transmit a new transmission request; and some of the idle sensors will sense data surpassing their threshold and transmit a new request for that data. The latter is dependant on the number of currently idle sensors and the probability that each of them detects an event.

This steady-state approach is well suited for application scenarios where the phenomenon sensed by each of the sensors is not heavily dependant on that sensed by the other, and where data is continuously transmitted to the gateway. In these scenarios the number of new application requests generated in an RRC can be modelled based only on the number of idle devices in that RRC. For applications where it is likely that data transmissions are sparse, and where a single environmental phenomenon is likely to meet the thresholds of multiple sensors, the model based on that for the 2C protocol where sensors do not receive new data until all conflicts are resolved is preferred.

We use the hybrid model described above to verify quasi-generic properties. We verify

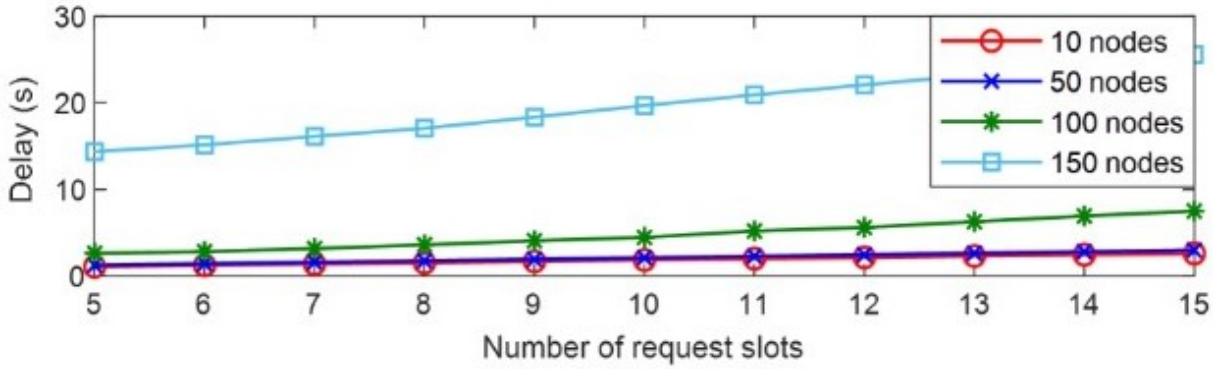


Figure 5.6: Ctrl-MAC data transmission times versus request slots comparisons. Image taken from [22].

properties such as

```
R<=9.5 [ F s1=8 ],
```

i.e. the expected time taken from the initial state to reach a state where $s1 = 8$ (i.e. the identified sensor has transmitted its data successfully) is less than 9.5.

The effect that the number of request slots has on the time that is needed from event detection to data transmission is investigated in [22]. This is achieved via the use of experiments performed on a reference Ctrl-MAC implementation on top of LoRa. The minimum number of request slots is declared to be five in order to meet the 10% duty cycle requirement, and experiments are run for a range of request slots. Figure 5.6 shows the average delay experienced by a sensor node in a Ctrl-MAC implementation with the specified number of request slots.

Model 5 allows us to verify properties about the time that a participating sensor device needs to transmit its data and the number of times it is going to be backed off before succeeding. We have created a suite of PRISM models with a varying number of request slots based on model 5. We then investigate the delays experienced by the individual sensor device. Figure 5.7 shows the results obtained by the model. Although the results produced by the model are much more constrained, the linear relationship between the number of request slots and the delay in message transmission is observed.

Our results support the simulation results obtained in [22]. The number of successful request slots in the equilibrium position of the protocol is linearly dependant on the total number of request slots in an RRC: the steady-state of the protocol maximises the number of successful requests, and this maximum increases as the number of possible successful request slots in an RRC increases. However, the number of successes is strictly less than the number of request slots, hence the ratio between the average number of successful transmission requests and the number of request slots is a proper fraction. As the durations of the request slots are the same, increasing the number of request slots from m_1 to m_2 would result in an $\frac{m_2}{m_1}$ times increase of the duration of the RRC (here we assume that the time needed by the gateway to broadcast the RRM

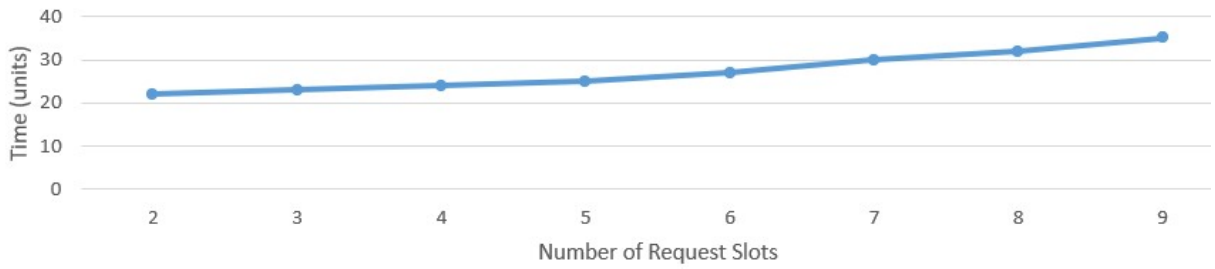


Figure 5.7: Ctrl-MAC data transmission times versus request slots comparison based on results from model 5. Based on a model with three sensor devices.

is negligible in comparison to the duration of time allocated for the request slots). Therefore, the increase in the average successful attempts will be strictly lower than the increase in the time needed for those attempts, and consequently, the throughput of the Ctrl-MAC protocol will decrease. We will discuss this further in Section 5.3.3.

5.3 Applying combinatorics

Currently, each sensor's choice of a request slot is modelled as a separate transition. As each sensor performs its choice uniformly at random, independently from other sensors, it could be desirable to perform those choices simultaneously. We are not interested in the order in which sensors pick their request slots, only at the final number of sensors choosing each slot. When applying counter abstraction to this model, we update the counters one sensor at a time when a sensor chooses a request slot (see Listing 5.6). That leads to a state space increase due to order symmetries. We attempt to apply a statistical model in order to resolve this issue. A statistical approach would not allow us to obtain exact verification results, but could allow us to obtain approximations and upper/lower bounds for properties we are interested in.

```
1 // each sending sensor chooses the slot they like
2 [] g=1 & idle=0 & sending>0 ->
3 1/5:(sending'=sending-1)&(send0'=min(send0+1,POPULATION))&(c0'=min(c0+1,2))+
4 1/5:(sending'=sending-1)&(send1'=min(send1+1,POPULATION))&(c1'=min(c1+1,2))+
5 1/5:(sending'=sending-1)&(send2'=min(send2+1,POPULATION))&(c2'=min(c2+1,2))+
6 1/5:(sending'=sending-1)&(send3'=min(send3+1,POPULATION))&(c3'=min(c3+1,2))+
7 1/5:(sending'=sending-1)&(send4'=min(send4+1,POPULATION))&(c4'=min(c4+1,2));
```

Listing 5.6: PRISM code for gateway module in model 3.

Consider the following example of 5 sensors and 5 request slots. We list a single scenario and the consequences for the values of the counters:

1. All five sensors start at idle.
   Counters: idle = 5

2. Three sensors detect an event and make requests during the same RRM.
   Counters: idle = 2, sending = 3.

3. One of the three sensors chooses at random to send on the second slot.
   Counters: idle = 2, sending = 2, send2 = 1.

4. Another of the three sensors chooses to send on the fourth slot.
   Counters: idle = 2, sending = 1, send2 = 1, send4 = 1.

5. The last sensor chooses to send on the second slot as well.
   Counters: idle = 2, sending = 0, send2 = 2, send4 = 1.

Notice that the order of steps 3, 4 and 5 has no effect on the final values of the counters; however, the model would consider the intermediate states for these different orderings as different states. This can cause a factorial increase in the size of the state space. The protocol is expected to be deployed on networks with a number of sensors much larger than the number of request slots: hundreds of sensors and a single-digit number of request slots. Therefore, the approach should be centred around the number of requests in each request slot rather than around the sensors. Statistical methods can be used to calculate the probabilities of different distributions of requests amongst the request slots at each iteration. This would allow steps 2 to 5 in the above example to be merged in our model. The goal is that this approach leads to further abstractions of the model, potentially removing the need to use PRISM at all. We base this approach on the population models inspired by mathematical biology presented in [86]. Figure 5.8 illustrates the idea behind this more abstract model. The number of counters used is significantly reduced as the previous families of states are now coalesced. Counters now model four possible sensor states:

- *idle*: Sensors that are in power-saving mode, having not yet sensed data above a given threshold.

- *sending*: Sensors that have sensed data above a threshold, and have synchronised with the gateway. These sensors will be making requests at this RRM.

- *back-off*: Sensors that have experienced a congested request slot, and are waiting to make another request.

- *done*: Sensors that have just made a successful request. These sensors will send their data during their allocated data slot on their allocated data channel before returning to power-saving mode.

This abstraction leads to transitions between counters that are more complex. Previously transitions were limited to either a single sensor at a time when sensors were performing a random choice, or to transferring all sensors in one counter to another when a time-dependant action

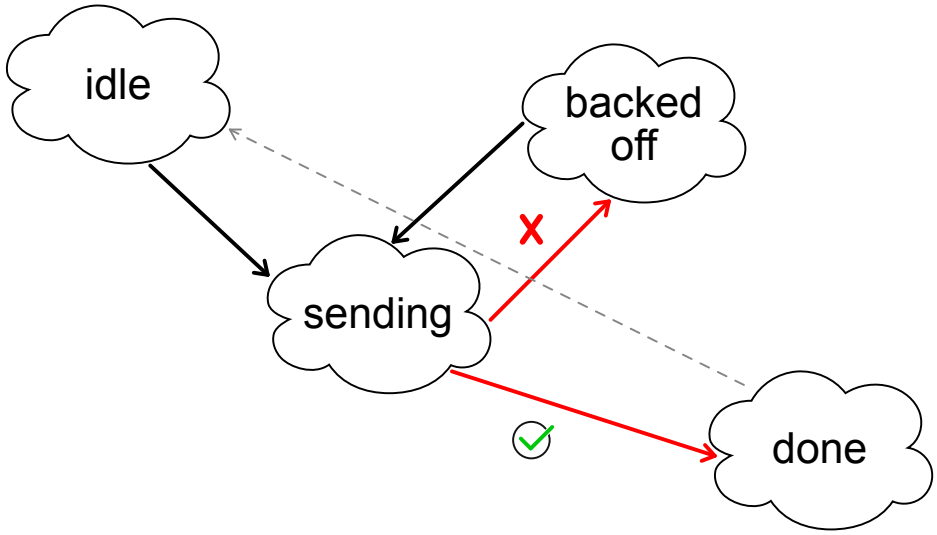


Figure 5.8: Counters for the model based on statistical analysis. Each cloud is a counter for a population, and arrows label possible ways in which sensors go from one state to another. The *sending* population has two outgoing transitions: to the *done* population for sensors that perform a successful transmission request (labelled by a green tick), and to the *backed off* population for sensors that experience congestion when sending their transmission requests. The dotted line represents the optional transition based on the type of environment that is being modelled.

occurred, such as lowering remaining back-offs during an RRM. The new more complex transitions need to transfer any number of sensors from one counter to another based on a probability defined using statistical analysis.

The probabilities of transitions from the *idle* state are dependant on the rate at which new data is sensed by the sensors. This will differ based on the data being sensed, the threshold, the environment, etc. This information can be gathered for each use case: a probability distribution can be created based on log files from the sensors. Similarly, transitions from the *done* to the *idle* state could be decided based on the use case. Sensors can stay in the *done* state for a constant time of one RRM if they are unable to sense while transmitting data, they might stay longer if the sensing period changes during the beginning of power-saving. It is also possible to absorb the *done* state into the *idle* state if the sensing pattern of sensors does not change during data transmission, i.e. devices that have successful transmission requests move from the *sending* state directly to the *idle* state. We proceed using the first of these options.

The transitions involving the *sending* counter only depend on its value: the request slots are chosen uniformly and at random, so the probability for the number of successful requests is dependant only on the total number of requests. I.e. for any pair (r,s) where r is the number of transmission requests and s is the number of request slots, there is a probability distribution $P(r,s)$ for the number of successful and unsuccessful transmission requests. Calculating this probability distribution will be further discussed in Section 5.3.1.

Similarly, the transitions from the *back-off* counter to the *sending* counter are also only dependant on the number of requests made in one RRC; however, these transitions are dependant

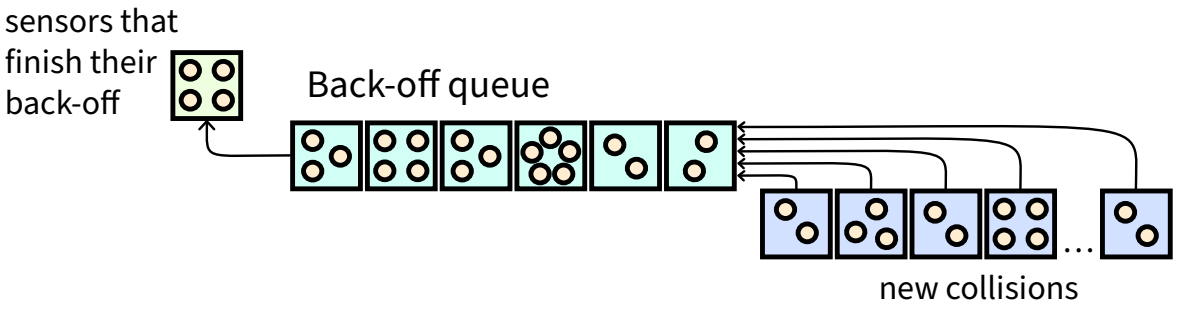


Figure 5.9: Viewing devices that have been assigned back-off (those tracked by the *back-off* counter) as a queue. Dequeued element on the left, enqueued elements on the right.

on the number of requests made in an RRC that has occurred in the past. As we have previously done, this can be accurately modelled by keeping track of the number of sensors with each available back-off duration (see Figure 5.9); however, this leads to an increase in the state space required for the models. We suggest that, by keeping track of only the total number of backed-off devices we can obtain an approximation of the number of devices that end their back-off periods. This would result in a loss of precision as the model would treat the back-off queue as a black box, and not keep explicit track of each backed-off sensor. This approach will be impractical for scenarios where environmental events are likely to trigger transmission requests from many sensor devices leading to bursty traffic; however, it is perfect for protocol implementations that reach a steady-state. In an equilibrium state RRMs in the past will not have a drastic difference from the current RRM, so we can approximate the previous ones based on the current one. This is supported by simulation results shown in Figure 5.10 where we see that a Ctrl-MAC implementation, whose sensors independently generate transmission requests. The simulation shows that the number of backed-off sensors, the number of idle sensors and the FTR value reach a stationary point. We continue this discussion in Section 5.3.1.

5.3.1 Obtaining the probability distributions

We briefly discuss the mathematics behind calculating the probability distribution for the number of successful transmission requests based on the number of request slots and the number of transmission requests received in an RRC. Consider a set of sensors S controlled by the Ctrl-MAC protocol. At each RRC, sensors that are ready to send information can choose a request slot from a set of request slots R. These sensors form a subset $S' \subset S$. An *allocation* $\nu_{n',r}$ is an r-tuple $\nu_{n',r} = \{i_1, i_2, i_3, ..., i_r\}$ where $n' = |S'|$ is the number of sensors attempting to send at this RRC, $r = |R|$ is the number of request slots, and i_j is the number of sensors that have chosen the j-th request slot. For clarity, we can refer to an allocation as $\nu_{n'}$ when the number of request slots is clear from the context.

Allocations have a few basic properties.

- For every j, $i_j \geq 0$, i.e. each request slot can be chosen by a non-negative number of

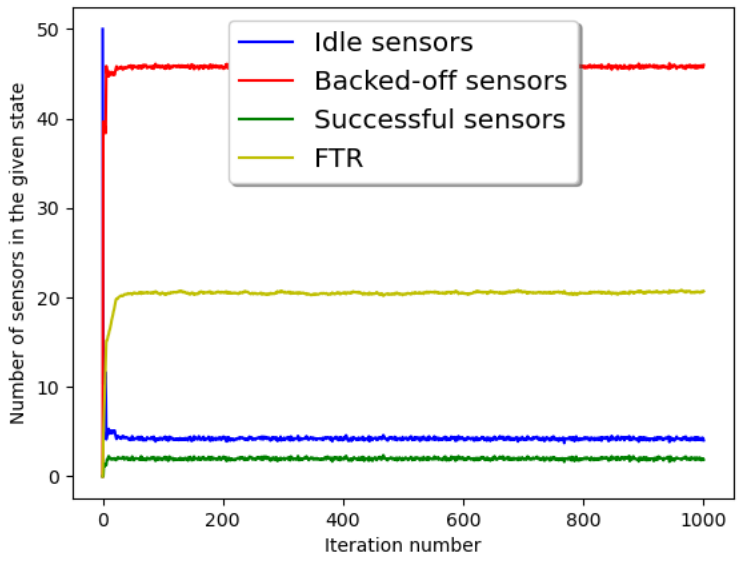


Figure 5.10: Simulation results showing the number of sensors, and FTR value in each state.

sensors.

- For every allocation $\nu_{n',r}$, $\sum_{j=0}^{r-1} i_j = n'$, i.e. each sensor that is ready to send must choose exactly one slot for the Request-Reply cycle.

- We will call the set of all possible allocations for given n' and r the allocation-set and label it as $N_{n',r}$. Similarly to allocations, the second index can be omitted for clarity.

First we consider the request slot choice phase to reason about the probabilities of collisions at each RRC (i.e. transitions from `sending` to `backed-off` and `done`).

Consider a Ctrl-MAC protocol configured with r request slots. Let n' sensors compete for a slot in a given Request-Reply cycle (note that this is different from n the total number of sensors as not all sensors will desire to send on each cycle).

Every allocation $\nu_{n',r}$ can be viewed as a $(n'+r-1)$-letter word made of an alphabet consisting of two symbols, say 'N' and '/' by placing 'N' i_j times for each element of $\nu_{n',r}$, and separating them using instances of '/'. Conversely, a word can be interpreted from left to right: starting at the first request slot, symbol 'N' signifies that there is 1 sensor that has chosen this request slot, and symbol '/' labels the change to the next request slot. For example, the word 'N/NN/NNN//N' corresponds to the allocation $\nu_{7,5} = \{1,2,3,0,1\}$. This is an RRC in which 7 sensors have made requests on 5 requests slots. There is 1 sensor requesting the first and the last request slot, 2 sensors requesting the second request slot, 3 sensors requesting the third request slot, and no sensors requesting the fourth request slot. Using this system, every word corresponds to an allocation and every allocation can be represented by a word. Therefore, there is a bijection between the set of words consisting of n' 'N' symbols and r '/' symbols and the

allocation-set $N_{n',r}$. It follows that both sets have the same cardinality. Therefore, we obtain a formula for the cardinality of an allocation-set:

$$|N_{n',r}| = \binom{n'+r-1}{n'} = \binom{n'+r-1}{r-1} = \frac{(n'+r-1)!}{(n')!(r-1)!}$$

Ctrl-MAC's behaviour is dependant on the number of sensor nodes that have chosen the same request slot. It is desirable to be able to reason about the probabilities of exactly x sensors choosing the same request slot. These are all of the allocations that have x as one of their entries. A way of doing this is to separate x sensors and 1 request slot, and find the number of allocations for the remaining $(n'-x)$ sensors on $(r-1)$ request slots. This can be done in the same manner as above.

$$|\{N_{n',r} \mid \exists j, i_j = x\}| = \binom{n'-x+r-2}{n'-x} = \binom{n'-x+r-2}{r-2} = \frac{(n'-x+r-2)!}{(n'-x)!(r-2)!}$$

We make an important observation about this formula. For an instance of the Ctrl-MAC protocol the number of request slots r remains fixed over time; however, the number of nodes that are ready to send at a given Request-Reply cycle n' will vary from cycle to cycle. We observe that:

$$|\{N_{n',r} \mid \exists j, i_j = x\}| = \binom{n'-x+r-2}{n'-x} = |\{N_{n'+1,r} \mid \exists j, i_j = x+1\}|$$

Therefore, to find the probability $P_C(x,n',r)$ of x sensors choosing the same request slot (and experiencing a collision), we can divide the number of allocations having x sensors choosing the same request slot by the total number of possible allocations. We omit some of the arguments, when the number of request slots and total number of requests received is clear from the context.

$$P_C(x,n',r) = \frac{(n'-x+r-2)!}{(n'-x)!(r-2)!} \times \frac{(n')!(r-1)!}{(n'+r-1)!}$$

$$= \frac{(n'-x+r-2)!(n')!(r-1)}{(n'-x)!(n'+r-1)!}$$

$$= \frac{(n')(n'-1)...(n'-x+1) \times (r-1)}{(n'+r-1)(n'+r-2)...(n'-x+r-1)}$$

Next, we consider how sensors that are backed off rejoin the population of sensors that is attempting to send a transmission request (i.e. transitions from *backed off* to *sending*). As discussed previously, keeping track of how many sensors have been backed off for a specific number of RRMs using a separate counter for each value (as in Figure 5.3) results in a complex model. Furthermore, the abstraction for the collisions as proposed above would be ill-suited for models that use an array of counter variables (one for each possible back-off duration). Using

a single counter variable instead prevents us from exactly modelling the states of all backed off devices, but can allow us to obtain an approximation for the behaviour of Ctrl-MAC which would allow us to reason about upper bounds on the delays experienced by the sensor nodes.

5.3.2 Implementation of Statistical approach

We have not yet implemented this approach, as in this thesis we focus on our range of PRISM models and their relative expressiveness and applicability. However, in Figure 5.8 we illustrate how the approach could be implemented, and leave the implementation as future work. We view the sensor devices participating in an instance of the Ctrl-MAC protocol as belonging to one of four populations depending on the state they are in. The sensor devices transfer from one population to another based on the probability distributions as described above. Devices in the *idle* state are currently sensing the environment, and when the sensed data meets a predefined threshold they change to the *sending* population. The rate at which this happens depends on the environment that is being sensed and will change between protocol instances. This one is impossible to be defined exactly as we do not know what phenomena will occur in the environment in the future, but can be approximated based on previous behaviour of the model. The devices in the *sending* population can either change to the *done* population or the *backed off* population based on whether the transmission request sent by the sensor device was successful or not. This probability depends on the number of requests sent during a given RRC and the number of request slots of the Ctrl-MAC implementation. We show how we can reason about this probability in Section 5.3.1. The probability of the transition from the *backed off* population to the *sending* population is the most complex as it is based on transmission requests from a number of RRCs in the past. We propose an approximation for this transition probability in Section 5.3.1. It can be based on the number of sensors that are currently in the *back off* population or based on the *sending* population (if we are considering equilibrium behaviour). The transition between the *done* and *idle* populations is straightforward as we expect sensor devices that have transmitted successful transmission requests to be able to transmit their data within one RRC and to be able to generate new transmission requests immediately afterwards.

5.3.3 Optimal number of requests

Using the mathematical formulas introduced in Section 5.3.1 we can obtain expressions for the probability that there is a specified number of successful transmission requests. We define $P_S(y, n', r)$ as the probability that there are y successful requests when there have been n' transmission requests during the same RRC of a protocol implementation with r request slots. For example, a 'collision' of exactly one sensor device in one request slot is actually that sensor

device having transmitted a successful transmission request.

$$P_S(1,n',r) = P_C(1,n',r)$$

We can then use this formula to obtain expressions for the expected number of successful requests $E_S(n',r)$ depending on the number of requests received and the number of request slots.

$$E_S(n',r) = \sum_{y=1}^{r} P_S(y,n',r) \times y$$

Table 5.7a shows the expected number of successful requests for a range of numbers of transmission requests received on a protocol implementation with five request slots. Notice that the number of successful requests increases as the number of submitted requests increases, reaches a maximum when the number of submitted requests is equal to the number of available request slots, and then decreases as the number of received requests increases further. Table 5.7b shows how that maximum value increases as the number of request slots increases. In fact, we notice a linear increase of about 0.36 successful requests for each additional request slot. This leads us to believe that increasing the number of request slots is beneficial, as a linear increase in the duration of RRCs results in a linear increase in the number of successful requests which decreases the relative overhead of sending RRMs. However, we must also take into account the other effects of increasing the number of request slots. As each request slot is of a fixed length, by increasing the number of request slots, we also increase the duration of each RRC. This impacts environments with bursty traffic negatively as it is less likely that a sequence of transmissions caused by the same event occurs across multiple RRCs. As previously discussed, the Ctrl-MAC protocol reaches its most efficient performance when the number of requests received during an RRC is low. Additionally, we notice that the base value for expected successful requests $E_S(2,2)$ is 1.00 which is larger than the subsequent linear increases. This offsets the overhead needed for the generation and transmission of RRMs. Lastly, we keep in mind that a sensor must wait for at least one RRM before it is successful and gets a data slot allocated. Longer RRCs would increase the time needed for the best case.

5.4 Summary

In this chapter we introduced the Ctrl-MAC protocol and the steps we took towards applying formal verification to it. We corrected all of the mistakes and ambiguities in the Ctrl-MAC specification, so that it accurately describes the design of the communication protocol. We provide our own description of the Ctrl-MAC protocol with a focus on the sensing and data transmission phase which is the phase we focused our verification efforts on.

We presented a suite of five PRISM models which we incrementally developed for the Ctrl-

n'	Backed off	Collisions	$E_S(n',5)$
1	0	0	1
2	0.40	0.20	1.60
3	1.08	0.52	1.92
4	1.95	0.90	2.05
5	2.95	1.31	2.05
6	4.03	1.72	1.97
7	5.16	2.12	1.84
8	6.32	2.48	1.68
9	7.49	2.82	1.51
10	8.66	3.12	1.34
11	9.82	3.39	1.18
12	10.97	3.63	1.03
13	12.11	3.83	0.89
14	13.23	4.01	0.77
15	14.34	4.16	0.66
16	15.44	4.30	0.56
17	16.52	4.41	0.48
18	17.59	4.50	0.41
19	18.66	4.59	0.34
20	19.71	4.65	0.29

(a) Number of sensors backed off, number of congested request slots, and number of successful requests depending on the number of transmitted requests in an RRC on a protocol with 5 request slots.

r	Optimal n'	$E_S(n',r)$
2	2	1.00
3	3	1.33
4	4	1.69
5	5	2.05
6	6	2.41
7	7	2.78
8	8	3.14
9	9	3.51
10	10	3.87
11	11	4.24
12	12	4.61
13	13	4.98
14	14	5.34
15	15	5.71
16	16	6.08
17	17	6.44
18	18	6.81
19	19	7.18
20	20	7.55

(b) Optimal average number of successful transmission requests in one RRC and the number of transmission requests associated with that RRC for varying number of request slots.

Table 5.7: Number of successful requests based on the number of request slots and the number of transmission requests sent in the same RRC.

MAC protocol. The first was based on a naive and straightforward approach and was instrumental in discovering inconsistencies in the protocol specification. While the model was limited in its scalability, the PRISM emulator allowed us to compare the behaviour of the protocol against basic use cases based on the performance desired by the protocol developers. For our second model we exploited our deeper understanding of the protocol to optimise our model by removing redundant variables and commands which were unnecessary for the modelling of the sensing and data transmission phase of the protocol. This was the model that most closely represented the operating principles followed by Ctrl-MAC. For our next models, we realised that the properties we wished to verify did not require this high level of detail. The third model we created used a judicious rearranging of the order of commands in order to remove the need to use a number of local variables. In practice, such a rearrangement is impossible due to the nature of asynchronous communication between the protocol gateway and the sensor devices; however, this had a significant impact on model performance. Additionally, Ctrl-MAC does not distinguish participating sensor devices in terms of priority, so our models also do not need to distinguish

between them. As we are interested in properties related to any sensor device, rather than an individual one, we investigated the application of symmetry reduction to Ctrl-MAC.

The final two models we created used counter abstraction to achieve symmetry reduction. The first of them had counter abstraction applied to all sensor devices. While this allowed a drastic increase in model sizes, it could only applied to *generic* properties - i.e. properties in which no individual device is identified in the property. The second abstract model resolved this issue by applying counter abstraction to all but one of the participating sensor devices. Doing so lessened the impact of symmetry reduction, but allowed for the verification of quasi-generic properties: i.e. properties in which an individual device is referred to.

We presented a range of properties for which we have verified our models and compared the effectiveness of each approach. We showed that with no symmetry reduction we can verify Ctrl-MAC models up to ten sensor devices. For our symmetry-reduced models we can verify the quasi-generic properties for twice that number of devices. Furthermore we can verify the generic properties for up to 40 sensors. In all cases our results support the experimental results produced by the protocol developers, which were derived via simulations of Ctrl-MAC instances.

Lastly, we discussed a statistical approach based on the the counter abstraction techniques that we applied. We proposed a simple mathematical model of the Ctrl-MAC protocol inspired by a population based approach in mathematical biology. The abstraction for the approach distinguishes three types of sensor devices: those that have been backed off after a collision, those that are actively attempting to send a transmission request and those that are currently sensing the environment for new data. We formulated the probability distributions associated with the transitions between each of these populations.

Chapter 6

GRIP - state of the art and new contribution

In this chapter we introduce the GRIP symmetry reduction tool - an existing PRISM package, and some improvements we have made to GRIP. We refer to the current version of GRIP, available from the PRISM website before we implemented any improvements, as GRIP 2.0; a public version in which we have implemented some fixes to GRIP and included some minor improvements as GRIP 2.1; and our proposed version, which can be used for specifications with synchronised transitions commands as GRIP 3.0. The results achieved in the development of this version have been published in [225]. In Section 6.1 we introduce GRIP. In Section 6.2 we give an overview of the process GRIP uses to obtain a symmetry-reduced specification. In Section 6.3 we describe how the symmetry reduction process can be extended to include synchronised transition commands. We propose a new version of GRIP, GRIP 3.0, which implements these extensions in Section 6.4.

6.1 Introduction

GRIP (Generic Representatives In PRISM) is a symmetry reduction tool for the PRISM model checker. Generic representatives were introduced by Emerson, Trefler and Wahl as a way to combine symbolic representation via BDDs and symmetry reduction in non-probabilistic model checking. GRIP was first introduced in [57] as a prototype tool that implements the extension of the generic representatives approach to probabilistic systems (DTMCs, Continious Time Markov Chains (CTMCs) and MDPs), which are represented via MTBDDs. GRIP version 2.0 was later released [65], introducing support for multiple local state variables, global variables, expressions over these variables, and multiple non-symmetric modules additional to the group of symmetric ones. Multiple local variables meant that it was easier to create modules with a large number of local states. Consequently, as each counter variable corresponds to a local state, reduced specifications have a large number of counter variables. This results in an increase of the sizes

of MTBDDs, so two new optimisation techniques were introduced to counteract this effect. The first allows for the elimination of a single counter variable based on the observation that the sum of the counter variables stays constant (and is equal to the number of symmetric modules). The second involves the use of local reachability analysis to remove unreachable states. A counter variable representing an unreachable state would have a constant value as there are always zero symmetric modules in an unreachable state. Therefore, the translated specification can omit that counter variable without any impact on the overall model, which can lead to an impressive reduction in the state space of the reduced model. Despite these new features it was still difficult to apply GRIP to complex models due to restrictions based on the modelling language. Specifications defined in Symmetric Probabilistic Specification Language (SPSL) [58] are guaranteed to be suitable for the generic representatives technique.

There are a few important differences between GRIP and an alternative symmetry reduction tool for PRISM, PRISM-symm. PRISM-symm is well suited for model specifications that consist of a small number of complex modules, while GRIP excels at large numbers of relatively simple modules. Our Ctrl-MAC example seems ideally suited to the use of GRIP. It consists of a (potentially large) number of fairly simple symmetric sensor devices sending their data via a (non-symmetric) gateway device. The features introduced in GRIP 2.0 would allow the gateway device to be modelled as an asymmetric module alongside a family of symmetric modules for the sensor devices. However, contrary to PRISM-symm, GRIP does not support synchronisation labels as a means of interaction between modules. Due to the nature of Ctrl-MAC's request-reply cycle, modules must synchronise to be capable of detecting network congestion. We have extended GRIP with support for synchronisation labels. We will describe this process in Section 6.3.

6.2 Current state of GRIP

The most recent version of GRIP (as of April 2023) and its source code are available from the PRISM webpage [97] together with a suite of seven example specifications and their reduced counterparts. There is no GRIP version number available on the webpage but the features of the tool are consistent with those of version 2.0 described in [65]. The tool works out of the box and is able to produce a basic translation of all example specifications; however, the source code (on Github [96]) indicates that there have been no changes made in the last 9 years. As such, all of the models produced by running GRIP on the example inputs and the example outputs available on the website result in errors when built using the latest version of PRISM. Additionally, when the local reachability analysis optimisation is applied to a specification it produces a malformed output. These errors are assumed to be a result of changes implemented in PRISM since GRIP's release, although there is nothing in PRISM's changelog that points to updates that would cause these errors.

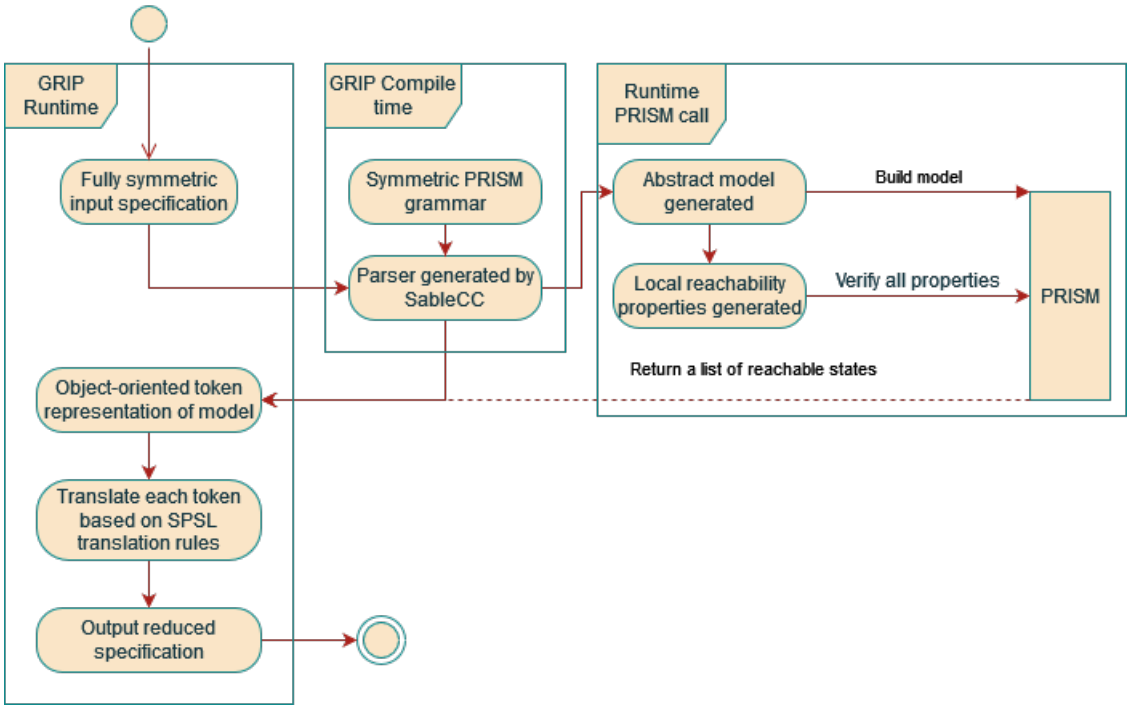


Figure 6.1: GRIP workflow process including source code compilation and specification translation. The runtime PRISM call in the right hand box is optional when the -optimise optimisation flag is set.

To discuss how we have updated and improved GRIP, we first give an overview of how GRIP worked before our changes. Figure 6.1 shows how the workflow process of GRIP is structured. When the source code for GRIP is compiled, a parser is created using the SableCC parser generator [84]. This parser is based on the grammar of *Symmetric PRISM* (contained in the `sp.grammar` file of the GRIP 2.0 release) and is used to create an object-oriented token tree representation of the input specification. At this stage during runtime, there is an optional system call to PRISM. If the `-optimise` flag is set, then GRIP performs local reachability analysis. GRIP generates an abstract specification based on the input and a set of properties that will be verified by PRISM (this process is explained in greater detail in Section 6.2.1). The result of this verification process is used to trim the object-oriented token tree representation produced by the parser. Once such a token representation is obtained, each token is translated individually based on the corresponding SPSL translation rule. These translations are then gathered together and output as the reduced specification produced by GRIP.

```
1 mdp
2 // shared coin
3 global counter : [0..48] init 24;
4
5 module process1
6   // program counter
7   pc1 : [0..3];
8   // local coin
```

```
9   coin1 : [0..1];
10
11  // flip coin
12  [] (pc1=0) -> 0.5 : (coin1'=0) & (pc1'=1) + 0.5 : (coin1'=1) & (pc1'=1);
13  // write tails -1  (reset coin to add regularity)
14  [] ((pc1=1)&(coin1=0)) -> (counter'=max(counter-1,0)) & (pc1'=2) & (coin1'
     =0);
15  // write heads +1 (reset coin to add regularity)
16  [] ((pc1=1)&(coin1=1)) & (counter<48) -> (counter'=counter+1) & (pc1'=2) &
      (coin1'=0);
17  // check
18  // decide tails
19  [] (pc1=2) & (counter<=8) -> (pc1'=3) & (coin1'=0);
20  //decide heads
21  [] (pc1=2) & (counter>=40) -> (pc1'=3) & (coin1'=1);
22  // flip again
23  [] (pc1=2) & ((counter>8) & (counter<40)) -> (pc1'=0);
24  // loop
25  [] (pc1=3) -> (pc1'=3);
26 endmodule
27
28 module process2=process1[pc1=pc2,pc2=pc1,coin1=coin2,coin2=coin1] endmodule
29 ...
```

Listing 6.1: Example input specification for GRIP based on a randomised consensus protocol using shared memory [10]. Module renaming is shown only for one symmetric module `process2`.

Input specifications for GRIP must be written in Symmetric PRISM (henceforth SP) in order for the translation process to be applicable. SP is a subset of the PRISM modelling language which allows the specification of programs with a degree of symmetry. Its syntax is analogous to the Symmetric Probabilistic Specification Language defined in [58]. An example input specification for GRIP can be seen in Listing 6.1. Table 6.1 lists the core syntax of SP as it appears in the latest version of GRIP. This includes the updates made when introducing the new features of version 2.0: SP supports asymmetric modules, global variables, multiple local variables and synchronisation labels. Note that the term 'number' denotes a numeric literal and that the term 'name' is an alphanumeric string used as the name of a module, variable or a synchronisation label. The element *arithmetic_expr* is an expression which evaluates to an integer value.

An SP specification consists of a model type declaration, optional global variable declarations, a number of modules, at least one of which must be symmetric, and at least one copy of that symmetric module. Each module has a name identifier, any number of local variable declarations, and at least one command statement. Module naming convention requires the names of asymmetric modules to contain no digits, while the names of symmetric modules must end with '1' (i.e. the digit one). Renamings for the family of symmetric modules increment this number.

spec	*type global_variable* module+ renaming+* | *constant_declaration* pctl+*
type	`nondeterministic` | `probabilistic` | `stochastic` | `mdp` | `dtmc` | `ctmc`
global_variable	`global` *variable*
module	`module` name *variable* statement+* `endmodule`
statement	[name?] *local_guard* & *global_guard* -> *stochastic_update* ;
stochastic_update	$\lambda_1 : assignment + \lambda_2 : assignment + ... + \lambda_n : assignment$ $(n > 0, \lambda_i \in [0,1], \sum \lambda_i = 1)$
variable	name : [number .. number] *init*? ;
assignment	(*atomic_assignment*) | (*atomic_assignment*) & *assignment*
atomic_assignment	name '= *arithmetic_expr*
renaming	`module` name = name [*identifier_renamings*] `endmodule`
identifier_renamings	*identifier_renaming* | *identifier_renaming* , *identifier_renamings*
identifier_renaming	name_i = name_j (for $i, j \in \mathbb{N}; i \neq j$)
constant_declaration	`const` *basic_type* name *init*? ;
basic_type	`int` | `double`
init	`init` number

Table 6.1: Grammar of Symmetric PRISM. PCTL-specific syntax is omitted.

So, for example,

```
module sensor2 = sensor1[s1=s2,s2=s1] endmodule
```

is the first renaming of a symmetric module `sensor1` with one local variable `s1`. Accepted model types are DTMCs, CTMCs, and MDPs, and similarly to PRISM, GRIP supports multiple keywords for each type for backwards compatibility (e.g. an MDP can be declared by both `mdp` and `nondeterministic`).

A module statement consists of an optional synchronisation label, a compound guard and a stochastic update to any of the local variables of that module and any global variable. As with PRISM, updates to global variables are not permitted if a synchronisation label is present. The compound guard is a conjunction of *local_guard* and a *global_guard*, the former being an expression over the local variables of the module, while the latter is an expression over either global variables or all copies of a local variable. An update has the form

$$\lambda_1 : a_1 + \lambda_2 : a_2 + ... + \lambda_n : a_n$$

where the λ_i are expressions giving a probability value, the a_i are updates to the values of one or more variables, and '+' is read as 'or' and represents a stochastic choice between these options as per the usual PRISM syntax described in Section 3.5.1. Property specifications consist of optional constant variable declarations and at least one PCTL property definition.

Note that SP grammar does not allow constants to be declared outside of property specifications, meaning that the common practice of using constants in model specification is not currently supported by GRIP. Furthermore, the constant and variable types supported do not include the boolean type, which is a valid option in the PRISM language. This is a minor concern as a boolean type can be easily substituted by an integer variable bounded between 0 and 1. We also note that despite the probabilities λ_i being correctly defined in the grammar, GRIP's parser encounters an error when a probability is represented as a fraction instead of a decimal value.

In order to translate an SP program to a reduced one, GRIP parses the input specification and creates an object-oriented tokenised representation. The parser is generated based on the SP grammar using the SableCC project. The associated syntax tree is used to initialise a Translator object. The Translator performs a walk on the syntax tree to extract the necessary model information, such as global variables, local variables, synchronisation labels, module names, module states, renamings, etc. A series of method calls are then executed that produce translations for each element of the reduced model: model type declarations, global variables, non-symmetric modules, formula definition (if the -eliminate flag is set), and the reduced generic process module. Most of these are generated in a relatively simple and straightforward manner as shown in Figure 6.2. The model type of the two specifications is the same, and the global variables and non-symmetric module declarations are directly copied from the original model.

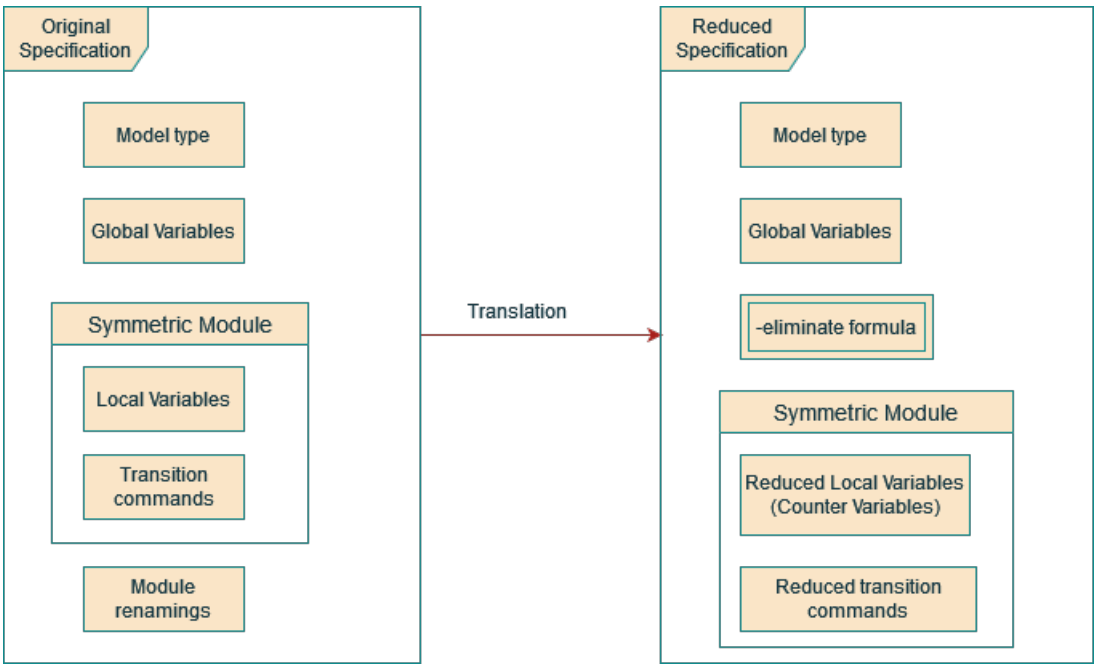


Figure 6.2: Visualisation of the translation steps of the current GRIP algorithm. Entities with the same name in the two specifications are direct copies. The double border around "`-eliminate` formula" denotes that this item is optional. We include a family of symmetric modules but no asymmetric modules (for simplicity).

The `-eliminate` formula is a single PRISM formula that is used to reduce the state space of the output specification by eliminating a single counter variable. This is done by replacing it with an expression for the sum of the remaining counter variables. This makes use of the invariant that the sum of all counter variables equals the number of symmetric modules (see Lemma 1).

Translating the symmetric modules into a single generic process is more complicated. This is done in two steps: translation of the local variables and translation of the commands of the module. The local variables are substituted by counter variables for each state a symmetric module can be in. Each counter variable keeps track of how many symmetric modules are in the state associated with it. Each transition statement of the original symmetric module is translated into one or more reduced statements which update the counter variables according to the original statement. Listing 6.2 shows the output produced by GRIP based on the *consensus* protocol specification from Listing 6.1. Lines 6 to 11 declare the counter variables replacing the local variables of the eight symmetric modules. Lines 13 to 20 are the translated transition statements. Note that each transition checks whether there is at least one symmetric module in a state associated with a particular counter variable. The update denotes the transfer of one module from one state to another by incrementing and decrementing the associated counter variables.

```
1  nondeterministic
2
3  global counter : [ 0 .. 48 ] init 24 ;
4
5  module generic_process
6   no_0 : [0..8] init 0;    // No modules in state (1,0)
7   no_1 : [0..8] init 8;    // No modules in state (0,0)
8   no_2 : [0..8] init 0;    // No modules in state (1,1)
9   no_3 : [0..8] init 0;    // No modules in state (3,0)
10  no_4 : [0..8] init 0;    // No modules in state (2,0)
11  no_5 : [0..8] init 0;    // No modules in state (3,1)
12
13    [] (no_1>0) -> 0.5:(no_1'=no_1-1)&(no_0'=min(no_0+1,8)) + 0.5:(no_1'=no_1
       -1)&(no_2'=min(no_2+1,8));
14    [] (no_0>0) -> (counter'=max(counter-1,0))&(no_0'=no_0-1)&(no_4'=min(no_4
       +1,8));
15    [] (no_2>0) & (counter<48) -> (counter'=min(counter+1,48))&(no_2'=no_2-1)
       &(no_4'=min(no_4+1,8));
16    [] (no_4>0) & (counter<=8) -> (no_4'=no_4-1)&(no_3'=min(no_3+1,8));
17    [] (no_4>0) & (counter>=40) -> (no_4'=no_4-1)&(no_5'=min(no_5+1,8));
18    [] (no_4>0) & ((counter>8)&(counter<40)) -> (no_4'=no_4-1)&(no_1'=min(no_1
       +1,8));
19    [] (no_3>0) -> true;
20    [] (no_5>0) -> true;
21
22 endmodule
```

Listing 6.2: Example output specification for a randomised consensus protocol using shared memory [10]. Output is generated by GRIP 2.0 based on the input from Listing 6.1

There is generally a one-to-one relationship between statements and translated statements. DTMCs are an exception where each statement is translated into a number of reduced statements equal to the number of symmetric modules present in the original specification. This is necessary to correctly model the fact that a counter variable with a higher value (i.e. more modules are in the state associated with it) is more likely to change in the next transition.

Each symmetric module must be in one and exactly one of the states corresponding to the counter variables. In addition, GRIP defines the same range for all counter variables in a given generic process, and that range is from 0 to the total number of symmetric modules. Combining these two observations we have the following lemma.

Lemma 1. *Let M be a set of symmetric modules and v a counter variable associated with M.*

(a) In all states, the value of v lies between 0 and $|M|$.

(b) At any state s, if the value of v is greater than zero, then for any v' associated with M with $v' \neq v$, the value of v' at s is less than $|M|$.

Name	Short description
(consensus)	a randomised consensus protocol using shared memory [10]
(byzantine)	an asynchronous Byzantine agreement protocol [33]
(rabin)	Rabin's n-process mutual exclusion protocol [197]
(fgf)	a simplified model of the Fibroblast Growth Factor signalling pathway [110]
(peer2peer)	a peer-to-peer protocol
(mutex)	a second mutual exclusion protocol by Pnueli and Zuck [191]
(leader)	and a minimum space shared memory leader election protocol [56]

Table 6.2: GRIP case studies.

This guarantees that no updates would actually result in out-of-range values, so we can apply the two fixes listed above without any effect on the underlying model. It was decided to extend the updates (i.e. adopt the second proposed fix), so that they explicitly cannot result in an out-of-range value instead of adding an additional expression to the guard. GRIP calculates what the new state and corresponding counter variable are only when translating the update part of a command. Making changes to a guard at this stage of the translation process would involve either backtracking, executing the same code multiple times, or implementing a major refactoring of GRIP's source code.

The seven example models presented on the GRIP website are described in Table 6.2. The example GRIP output for all seven models as well as the input specification for the *consensus* from Table 6.2 cannot be built by the latest version of PRISM. When executing GRIP 2.0 on *byzantine, mutex, leader,* and *peer2peer* we receive output identical to the one listed on GRIP's webpage; however, *consensus, fgf* and *rabin* produce a more complex reduced specification than expected. This is because these specifications were originally translated using both the *-eliminate* and *-optimise* optimisations; however, the latter results in a malformed generic module with no transition statements and only a single counter variable.

We have produced two updates to GRIP: version 2.1 and version 3.0. The former includes a number of updates that restore GRIP to its previous functionality, while the latter provides support for models employing synchronisation which we describe in Section 6.4.

6.2.1 Local reachability analysis optimisation

The local reachability analysis optimisation attempts to reduce the complexity of the output specification by reducing the number of counter variables. If a counter variable does not change its value through any of the model's transition commands, then it can be removed without impacting the correctness of the model. As transitions of the reduced specifications represent the change of a symmetric module from one state to another, a counter variable associated with a state that is never reached would not appear in any of the transition updates (and have zero as its value). Thus this optimisation focuses on identifying states of the symmetric modules that are unreachable.

In GRIP, local reachability analysis optimisation works by first creating an abstract model specification based on the input specification and then invoking PRISM to verify a set of properties. The abstract model contains only a single copy of the symmetric module with all global variables, asymmetric modules, and any expressions involving any of them removed. This is a simplified version of the input specification whose transition statements have their guards relaxed by the removal of expressions involving global variables. The idea behind the local reachability analysis optimisation is that if the symmetric module cannot reach a particular local state with these relaxed guards, it will also not be able to reach it with the original ones.

6.3 Synchronisation and Generic Representatives

Consider the following simple model that involves synchronisation. Two devices each call heads or tails for the flip of a coin. The two devices make their decisions at random and with equal probability, and must simultaneously reveal their choices, at which point they will terminate (we do not model any consequence of the coin toss). Figure 6.3 shows the state diagram of the system. All devices start at a state *0 - Initial state*, then they have equal probability to move to state *1 - chosen to call heads*, or state *2 - chosen to call tails*. Once the devices reach the last two states, they can move to state *3 - End state*. An example SP specification for this system is provided in Listing 6.3. Note that `si=j` represents that device *i* is currently in state *j*.

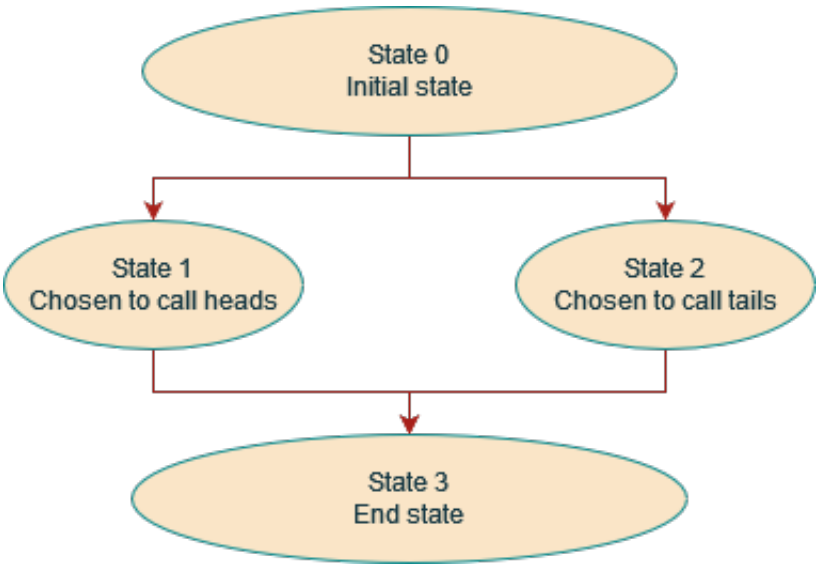


Figure 6.3: State diagram for the coin toss system.

```
1 dtmc
2
3 module device1
4   s1 : [0..3] init 0;
5
6   [] s1=0 -> 0.5 : (s1'=1) + 0.5 : (s1'=2);
```

```
7   [a] s1=1 -> (s1'=3);
8   [a] s1=2 -> (s1'=3);
9   [] s1=3 -> (s1'=3);
10
11 endmodule
12 module device2 = device1[s1=s2,s2=s1] endmodule
```

Listing 6.3: Example: coin toss with synchronisation

If we were to disregard the synchronisation label and run GRIP on this specification, we would receive the reduced specification shown in Listing 6.4 (but without the synchronisation labels). We have added the synchronisation labels to show which statements correspond to the two commands we want to synchronise.

```
1  probabilistic
2
3  module generic_process
4   no_0 : [0..2] init 2;    // No modules in state (0)
5   no_1 : [0..2] init 0;    // No modules in state (1)
6   no_2 : [0..2] init 0;    // No modules in state (2)
7   no_3 : [0..2] init 0;    // No modules in state (3)
8
9    [] (no_0>0) -> 0.5:(no_0'=no_0-1)&(no_1'=min(no_1+1,2))
10                + 0.5:(no_0'=no_0-1)&(no_2'=min(no_2+1,2));
11   [] (no_0>1) -> 0.5:(no_0'=no_0-1)&(no_1'=min(no_1+1,2))
12                + 0.5:(no_0'=no_0-1)&(no_2'=min(no_2+1,2));
13   [a] (no_1>0) -> (no_1'=no_1-1)&(no_3'=min(no_3+1,2));
14   [a] (no_1>1) -> (no_1'=no_1-1)&(no_3'=min(no_3+1,2));
15   [a] (no_2>0) -> (no_2'=no_2-1)&(no_3'=min(no_3+1,2));
16   [a] (no_2>1) -> (no_2'=no_2-1)&(no_3'=min(no_3+1,2));
17   [] (no_3>0) -> true;
18   [] (no_3>1) -> true;
19 endmodule
```

Listing 6.4: Example: reduced coin toss specification disregarding synchronisation

Let us examine lines 7 and 8 of the input specification (see Listing 6.3) and consider how the synchronisation affects the devices' behaviour. Without synchronisation labels, each device would be able to progress to state 3 (c.f. Figure 6.3) as soon as it enters state 1 or 2, whereas with synchronisation labels, each device must wait until the other device also reaches one of the states that satisfy the guard of a synchronised command. Furthermore, synchronisation requires all updates to be executed simultaneously. We can think of synchronisation labels as having two key effects: they tighten the guard of the statement as the behaviour of a module now depends on the other modules of this symmetric block, and they cause all updates to be executed simultaneously.

We now discuss the reduced specification produced by GRIP and the effect of synchronisa-

tion. In particular we examine lines 13 – 16 of Listing 6.4. As we have seen, these statements have been created by GRIP (without the synchronisation labels). The updates are simple: in each of them the value of the `no_3` counter is increased by one, i.e. a module moves to state 3. In two of them that module has arrived at that state from state 1, in the other two it has arrived from state 2. The guard of line 13 is equivalent to “there are 1 or 2 modules in state 1" and line 14 to “there are 2 modules in state 1" whereas line 15 and 16 are “there are 1 or 2 modules in state 2", and “there are 2 modules in state 2" respectively. Out of these, lines 14 and 16 would be acceptable even with synchronisation but the guards of the other two would need to be tightened. Only having one module in state 1 or 2 would not be sufficient but would also require the other module to be in the corresponding state. When exactly one module is in state 1, the other module would need to be in state 2 and vice versa. Listing 6.5 shows what a reduced version of the specification could look like if the guards were strengthened to accommodate this observation.

```
1  probabilistic
2
3  global total : [ 0 .. 2 ] init 0 ;
4
5  module generic_process
6   no_0 : [0..2] init 2;    // No modules in state (0)
7   no_1 : [0..2] init 0;    // No modules in state (1)
8   no_2 : [0..2] init 0;    // No modules in state (2)
9   no_3 : [0..2] init 0;    // No modules in state (3)
10
11    [] (no_0>0) -> 0.5:(no_0'=no_0-1)&(no_1'=min(no_1+1,2)) + 0.5:(no_0'=no_0
       -1)&(no_2'=min(no_2+1,2));
12    [] (no_0>1) -> 0.5:(no_0'=no_0-1)&(no_1'=min(no_1+1,2)) + 0.5:(no_0'=no_0
       -1)&(no_2'=min(no_2+1,2));
13    [a] (no_1=0) & (no_2=2) -> (no_2'=0)&(no_3'=min(no_3+2,2));
14    [a] (no_1=1) & (no_2=1) -> (no_1'=0)&(no_2'=0)&(no_3'=min(no_3+2,2));
15    [a] (no_1=2) & (no_2=0) -> (no_1'=0)&(no_3'=min(no_3+2,2));
16    [] (no_3>0) -> true;
17    [] (no_3>1) -> true;
18
19 endmodule
```

Listing 6.5: Example: reduced coin toss specification with synchronisation

Note that in this example, the updates for all synchronised commands in the output specification are the same. This might suggest that the three commands could be combined. This is true in this example but is not true in general. Synchronised statements are likely to carry out different updates. The reduced updates will therefore consist of a number of distinct assignments each necessitating a distinct command. For each synchronisation label, guard and update combination, we must consider the possible ways of allocating the symmetric modules between

the states accepted by the guards. These combinations are determined using the result of Lemma 2.

Lemma 2. *Let M be a family of symmetric modules with synchronised commands with $|M| = m$. Consider one block of synchronised commands that are all synchronised on the same action. For each synchronised command c, let $s(c)$ be the number of local states that satisfy its guard, and let k be the sum of the $s(c)$ over the synchronised commands for M. The number of unique reduced commands that must be generated for that block of synchronised commands is the number of weak compositions of m into k unique parts, $wc(m,k)$, where*

$$wc(m,k) = \binom{m+k-1}{k-1} = \binom{m+k-1}{m} \tag{6.1}$$

It follows that the number of reduced commands generated by the translation process depends on m and k and increases exponentially with those two values.

To add support for synchronisation to GRIP we must develop new SPSL translation rules for synchronised commands. The original rules, introduced in [58], are explained in Section 3.6.3.

Counter abstraction requires the creation of *counter variables* in the reduced specification. Each family of symmetric modules is replaced by a single generic module with $|S(M)|$ counter variables, each of which have range from 0 to $\#M$. These variables have the form `count_M_k` where each counter keeps track of the number of copies of M currently in state $f_M^{-1}(k)$ for $k \in [1,t]$. All counter variables are initialised to 0, except for `count_M_`$f_M(init(M))$ which is set to $\#M$.

We consider the translation of a synchronised command

$$[label] \quad \mathsf{local\text{-}expr}(M) \wedge \mathsf{symm\text{-}expr}(M) \rightarrow \mathsf{stoch\text{-}update}(M) \tag{6.2}$$

$$[a_1] \quad e_1 \wedge s_1 \rightarrow su_1.$$

We can view each synchronised command as a command of the form above without loss of generality because every guard of an SP statement can be separated into expressions over local variables ($\mathsf{local\text{-}expr}(M)$) and expressions including non-local variables ($\mathsf{symm\text{-}expr}(M)$). The latter would need to be fully symmetric in order for translation to be applied, and either of them can be taken as *true* should no expressions of that type be present.

Without synchronisation, GRIP would split the translation process into cases based on the local states satisfying the guard, one per $l \in \mathsf{SAT}_M(e_1)$ (see Fig. 6.4). Then, for each case, a separate reduced generic statement is generated using the following process. The $\mathsf{local\text{-}expr}(M)$ part of each guard is replaced with a condition `count_M_`$f_M(l) > 0$ (or in the case of DTMCs, each statement gets translated into a series of statements where `count_M_`$f_M(l) > i$ for $0 \leq i < \#M$ is used). This condition asserts that some copy of M has a local state required to satisfy the local part of the guard of this statement. Both the remainder of the guard $\mathsf{symm\text{-}expr}(M)$ and

the stochastic update stoch-update(M) are translated in the context of the state l, so that their reduced counterparts do not make use of any variables local to M. Updates affecting variables local to M are replaced with updates to two counter variables: if l is the local state considered in the current case, and l' is the local state reached by performing all local variable updates on l, then the resulting reduced update must decrement $\texttt{count\_}M\texttt{\_}f_M(l)$ (representing a module copy leaving state l), and increment $\texttt{count\_}M\texttt{\_}f_M(l')$ (representing a module copy arriving at state l').

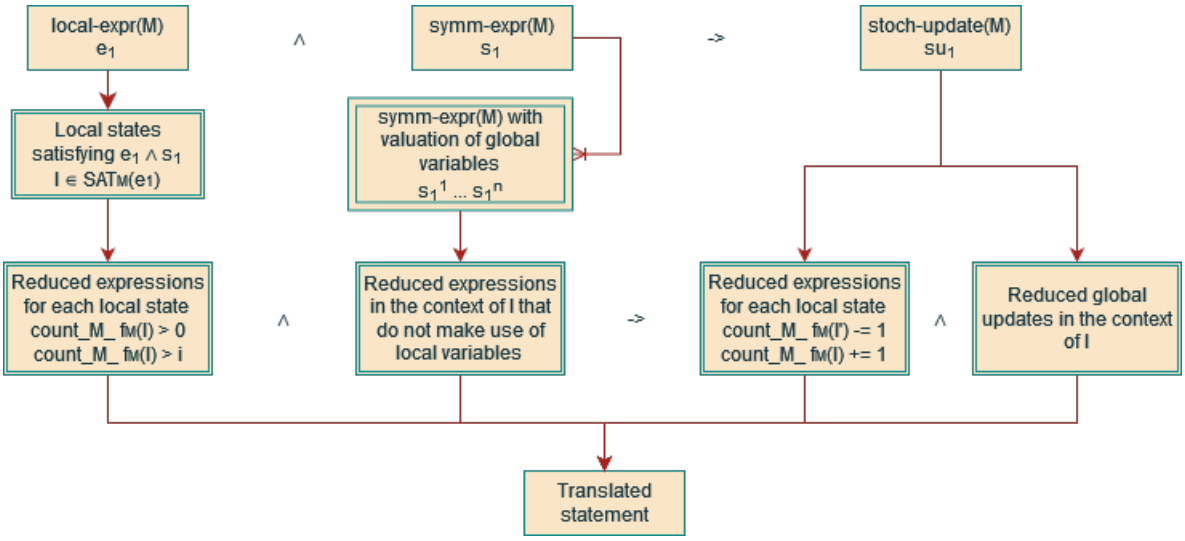


Figure 6.4: Translation of a non-symmetric statement.

We approach synchronised statements using a similar methodology (see Figure 6.5). First, we note that PRISM does not allow synchronised commands to perform updates to global variables, so all updates of synchronised statements will only change the local state of a module. For the purposes of illustration, we assume that all synchronised statements perform only a single update (i.e. of the form $p_1 : u_1$ where $p_1 = 1$) rather than a stochastic choice of updates. We will discuss translation of multiple updates later.

Translating statements with synchronisation differs from translating their non-synchronised counterparts in that we must consider all commands with a given label instead of being able to carry out the translation process one command at a time. Synchronisation creates coupling between commands in a synchronised block as the result on the overall state of the model of a module performing one command also depends on all other symmetric modules and the statements they execute.

Suppose a block of synchronised commands of the form denoted in expression (6.2), and repeated here for convenience:

$$[label] \quad \text{local-expr}(M) \wedge \text{symm-expr}(M) \rightarrow \text{stoch-update}(M) \tag{6.2}$$

with the same label is present in $\#M$ symmetric modules. For the symmetric block to be enabled,

each symmetric module must satisfy the guard of at least one synchronised statement. When the symmetric block is executed, each symmetric module executes one of the enabled commands, so it must be in a local state that satisfies the guard of that command. An example of such a synchronised block is shown in Listing 6.6. This block contains four synchronised commands, each of which performs a single update to one local variable. Each symmetric module must satisfy the guard of at least one of these commands in order for the synchronised block to be enabled, i.e. the value of the `s1` variable must be 1, 2, 3, or 4. We note that in general the guards do not need to be disjoint and there can exist local states that satisfy the guards of more than one command.

```
1 [a] s1=1 -> (s1'=5);
2 [a] s1=2 -> (s1'=6);
3 [a] s1=3 -> (s1'=7);
4 [a] s1=4 -> (s1'=8);
```

Listing 6.6: Example: a block of synchronised statements

For each command we can evaluate the local part of the guard, $\mathsf{local\text{-}expr}(M)$, to find which local states satisfy that guard. Let the collection of those states across all commands be $l^1, l^2, \ldots, l^z$ under some ordering. We note that the elements of this list may not be distinct as a given state l^i can satisfy the local guard of one synchronised statement while l^j could be the same state that satisfies the local guard of another synchronised statement. Let $w^1, w^2, \ldots, w^z$ be the number of symmetric modules that are in the corresponding local state. We can then use counter variables to describe the scenario by translating each expression e_i in the guards into the condition $\bigwedge_{i=1}^{z} \texttt{count\_}M\texttt{\_}f_M(l^i) = w^i$. We note that Lemma 1 enables us to omit every expression where $w^i = 0$ without impacting the end result, i.e.

$$\bigwedge_{i=1}^{z} \texttt{count\_}M\texttt{\_}f_M(l^i) = w^i \equiv \bigwedge \texttt{count\_}M\texttt{\_}f_M(l^i) = w^i$$

for $1 \leq i \leq z$ if $w^i \neq 0$.

The global part of the guards, $\mathsf{symm\text{-}expr}(M)$, is then translated by performing a conjunction of only those parts which belong to a statement that is being executed, and evaluated in the context of the corresponding local state l^i. For each local state l^i with $w^i > 0$, we translate the global guard of the statement whose local guard is satisfied by l^i. We then combine the translated global guards. The individual translation process here is the same as for the global part of a non-synchronised command. The translation of updates follows a similar idea: for a given assignment of $w^1, w^2, \ldots, w^z$, we find the states l^i with $w^i > 0$, and translate the updates of the corresponding commands.

The updates are again translated individually, as they would be if synchronisation was not present, but are afterwards combined based on the value of w^i. As mentioned above, in PRISM synchronised statements cannot make updates to global variables, so each update can only make

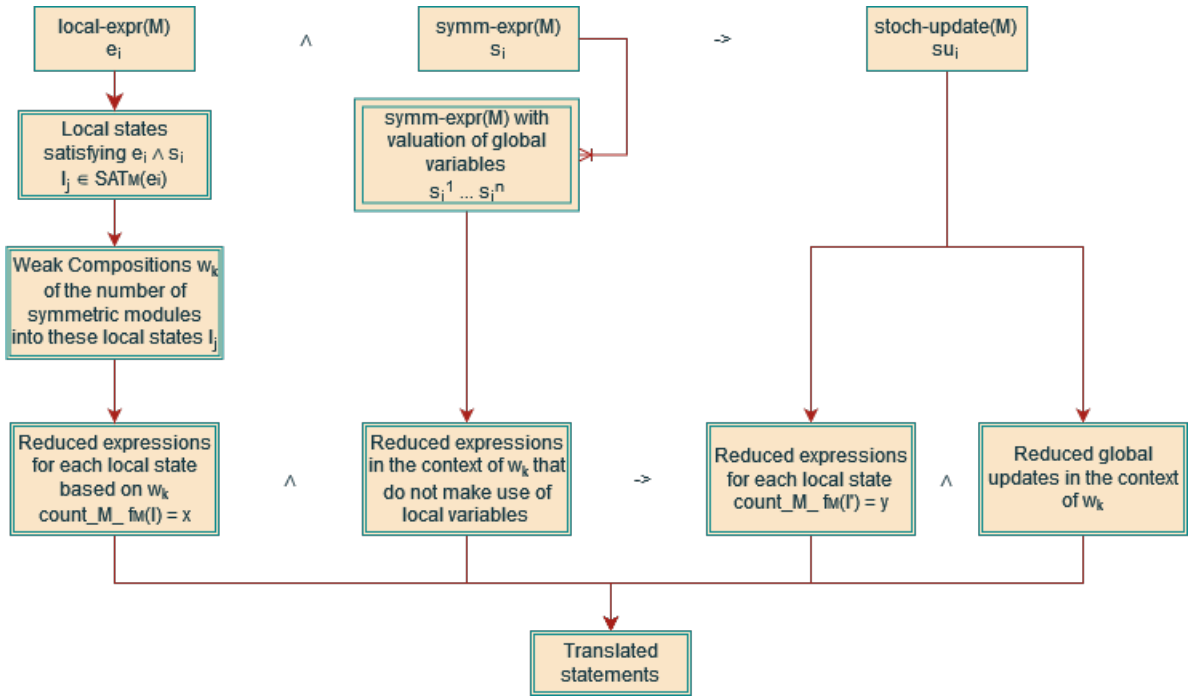


Figure 6.5: Translation of a block of symmetric statements.

changes to the local state of the module, which when reduced has the effect of only decrementing one counter variable and incrementing another. We need to take into account the number of symmetric modules that perform this update but if the command has a single update with probability 1, then all modules make the same changes to their local states. Therefore, the resulting translation would need to decrease one counter variable by the number of modules performing this update and increase another counter variable by the same value.

Although we have assumed that synchronised statements execute a single update, we will briefly discuss synchronised statements which involve a stochastic choice between multiple updates. Such updates are not needed for modelling the Ctrl-MAC protocol discussed in chapter 5, and so we are not considering them for the current GRIP update. Furthermore, if the PRISM examples involving synchronised statements listed on the PRISM webpage [193] are typical, then synchronised statements rarely involve stochastic updates. For updates of the form $p_1 : u_1 + p_2 : u_2 + \cdots + p_n : u_n$, the translation would need to enumerate all possible ways that the modules executing the statement can choose the updates they perform, and calculate the resulting probabilities. For example, the basic case of the stochastic update $p_1 : u_1 + p_2 : u_2$ with a guard satisfied by a single state l^x with w^x modules would result in the following binomial expansion:

$$\sum_{k}^{w^x} \left(\binom{w^x}{k} (p_1)^{w^x - k} (p_2)^k : \mathfrak{h}(u_1, w^x - k) \wedge \mathfrak{h}(u_2, k) \right)$$

where $\mathfrak{h}(u, v)$ denotes the changes in counter variables resulting from the execution of update u by v symmetric modules. This results in a $O(w^x)$ increase in the number of possible updates.

This generalises to

$$\sum_{k_1+\cdots+k_n=w^x} \left(\binom{w^x}{k_1,\ldots,k_n} (p_1)^{k_1}(p_2)^{k_2}\ldots(p_n)^{k_n} : \mathfrak{h}(u_1,k_1) \wedge \mathfrak{h}(u_2,k_2) \wedge \cdots \wedge \mathfrak{h}(u_n,k_n) \right)$$

which results in a $O(\binom{n+w^x-1}{n-1})$ increase in the number of updates. These translations would lead to a significantly more complex output specification, and would have an adverse impact on the performance gained by performing counter abstraction.

This concludes the process to generate a generic statement based on a synchronised statement block given an assignment of $w^1, w^2, \ldots, w^z$ of the symmetric modules into local states. As the behaviour of the synchronised block depends on the states of the symmetric modules prior to their execution, a different reduced statement must be generated for each possible assignment of $w^1, w^2, \ldots, w^z$.

We introduce and expand some notation to help formalise this process. The set of states $l^1, l^2, \ldots, l^z$ that a symmetric module can be in while satisfying the local guard of at least one synchronised statement is denoted

$$\mathsf{SAT}_M(\mathbf{e}) = \bigsqcup_{1 \le i \le r} \mathsf{SAT}_M(e_i)$$

where $\mathbf{e} = \{e_1, e_2, \ldots, e_r\}$ is the set of local parts of the guards of all r synchronised statements. We note again that the states in $\mathsf{SAT}_M(\mathbf{e})$ may not be distinct.

An assignment of the $\#M$ symmetric modules to this set of states is given by $\mathbf{w} = \{w^1, w^2, \ldots, w^z\}$ where $w^i \geq 0$ and $\sum_{i=0}^{z} w^i = \#M$. This is a weak composition of $\#M$ into z parts (corresponding to each of the $l^1, l^2, \ldots, l^z$ states) and we define

$$wc : \mathbb{R} \times S(M) \to S(M)^n$$

to be the function that generates all possible weak compositions with n being the total number of weak compositions. We introduce some shorthand notation: let

$$g_M : S(M)^z \to \{1, 2, \ldots, t\}^z$$

be the function obtained by applying f_M to all elements in the input of g. For example,

$$\texttt{count\_}M\texttt{\_}g_M(\mathsf{SAT}_M(\mathbf{e})) \equiv \{\texttt{count\_}M\texttt{\_}f_M(l^i) \mid i \in [1,z]\}$$

where both expressions denote the set of counters that correspond to the local states of M that enable the block of synchronised statements whose guards have $\mathbf{e}$ as their local parts.

Figure 6.6 lists new translation rules to be used in the translation of an SPSL specification $\mathscr{P}$ containing synchronised statements into a generic SPSL specification $\mathfrak{h}(\mathscr{P})$ where $\mathfrak{h}$ is the

translation function. These rules consider a whole family of synchronised commands rather than individual commands, as the reduced statements are generated based on multiple synchronised commands rather than a single one. Rules that are unchanged can be found in the original table in Figure 3.5.

When translating synchronised statements for a module M, we consider the fact that not all expressions in their guards and updates will be symmetric ones, but some will involve the local state of M. We will assume that the guards of the synchronised statements are of the form $e_i \wedge \mathsf{expr}(M)$, where e_i has the form $\mathsf{loc\text{-}expr}(M)$ and $\mathsf{expr}(M)$ is a symmetric expression. This can be done without loss of generality as a guard without a local part or a guard without a symmetric part can be expressed in this form by substituting the corresponding argument with *true*.

The translation process translates a synchronised block as a whole, on a case by case basis. We construct a collection of all local parts of the guards $\mathbf{e} = \{e_1, e_2, \ldots, e_r\}$ and use it to find all local states $l \in \mathsf{SAT}_M(\mathbf{e})$. Then we generate the weak compositions of $\#M$ into $|\mathsf{SAT}_M(\mathbf{e})|$,

$$\mathbf{W} = \{\mathbf{w}_1, \ldots, \mathbf{w}_n\} = wc(\#M, \mathsf{SAT}_M(\mathbf{e}))$$

and use those to split the translation process into cases, one for each $\mathbf{w} \in \mathbf{W}$. The worst case for the translation algorithm of a single synchronised block containing r statements is given by

$$O(\binom{r \times |S(M)| + \#M}{\#M}).$$

Similarly to the worst case complexity for translating an SPSL specification without synchronisation [58], in practice, the local part of the guards of transition statements is often strongly constrained, so the worst case complexity is rarely achieved. The worst case complexity for translating an entire specification that contains blocks of synchronised statements is easily derived by factoring in the number of synchronised blocks. However, it is generally the case that specifications involving multiple synchronised blocks will have one of those blocks larger and more complex than the rest. A small increase in the number of synchronised statements r results in a much larger increase in overall complexity due to the binomial coefficient in the formula. As such the worst case complexity for translation of the entire specification is mainly dependant on the worst case complexity of the largest block of synchronised statements.

For each weak composition $\mathbf{w}$ we examine the local states l^j for which $w^j > 0$ and the statements they correspond to. The e_i part of the guard of those statements is translated into the condition $\bigwedge(\texttt{count\_}M\texttt{\_}l^j = w^j)$, and the $\mathsf{expr}(M)$ part is translated in the context of l^j and combined with the other translated parts. The translation of the variable updates follows a similar process. Consider updates of statements corresponding to $w^j > 0$ of the form:

$$(v^1 := e^1) \| (v^2 := e^2) \| \ldots \| (v^t := e^t)$$

$\mathscr{P}$	$\mathfrak{h}(\mathscr{P})$
statement(M), where [a] is a label and e has form local-expr(M)	$\mathfrak{h}$(statement(M)), with $\mathsf{SAT}_M(\mathbf{e}) = \{l^1,\ldots,l^z\}$ and $\{\mathbf{w}_1,\ldots,\mathbf{w}_n\} = wc(\#M, \mathsf{SAT}_M(\mathbf{e}))$, $\mathbf{e} = \{e_1,\ldots,e_r\}$
[a]$e_1 \wedge$ expr(M_i) $\rightarrow$ stoch-update(M) ... [a]$e_r \wedge$ expr(M_i) $\rightarrow$ stoch-update(M)	[a]count_M_$g_M(\mathsf{SAT}_M(\mathbf{e})) = \mathbf{w}_1 \wedge \mathfrak{h}$(expr($M_i$), $\mathbf{w}_1$) $\rightarrow \mathfrak{h}$(stoch-update(M), $\mathbf{w}_1$) ... [a]count_M_$g_M(\mathsf{SAT}_M(\mathbf{e})) = \mathbf{w}_n \wedge \mathfrak{h}$(expr($M_i$), $\mathbf{w}_n$) $\rightarrow \mathfrak{h}$(stoch-update(M), $\mathbf{w}_n$)
stoch-update(M)	$\mathfrak{h}$(stoch-update(M), $\mathbf{w}$)
update(M)	$\mathfrak{h}$(update(M), $\mathbf{w}$)
update(M), where $v^j \in var(M)$ and e^j has form local-expr(M)	$\mathfrak{h}$(update(M), $\mathbf{w}$), where $\mathbf{w} = \{w^1,\ldots,w^z\}$ $\mathbf{l}' = \mathbf{l}[v^1 := eval(l,e^1),\ldots,v^t := eval(l,e^t)]$
skip $(v_i^1 := e_i^1) \| \ldots \| (v_i^t := e_i^t)$	skip (count_M_$g_M(\mathbf{l})$:= count_M_$f_M(\mathbf{l}) - \mathbf{w}$) $\|$(count_M_$g_M(\mathbf{l}')$:= count_M_$g_M(\mathbf{l}') + \mathbf{w}$)
expr(M_i), where e has form local-expr(M)	$\mathfrak{h}$(expr(M_i), $\mathbf{w}$)
e_i	$eval(l,e)$
symm-expr	$\mathfrak{h}$(symm-expr)
$\sum_{1\le j\neq i\le \#M} e_j$	$\sum_{m\in S(M)}(eval(m,e) * \texttt{count\_}M\texttt{\_}f_M(m)) - eval(l,e)$
$\prod_{1\le j\neq i\le \#M} e_j$	$\prod_{m\in S(M)}(eval(m,e)$**count_$M$_$f_M(m))/eval(l,e)$
$\bigwedge_{1\le j\neq i\le \#M} e_j$	$\sum_{m\in \mathsf{SAT}_M(e)}$ count_M_$f_M(m) = \#M$ (if $l \models e$) $\sum_{m\in \mathsf{SAT}_M(e)}$ count_M_$f_M(m) = \#M - 1$ (if $l \not\models e$)
$\bigvee_{1\le j\neq i\le \#M} e_j$	$\sum_{m\in \mathsf{SAT}_M(e)}$ count_M_$f_M(m) > 0$ (if $l \not\models e$) $\sum_{m\in \mathsf{SAT}_M(e)}$ count_M_$f_M(m) > 1$ (if $l \models e$)
expr(M_i) $\bowtie$ expr(M_i)	$\mathfrak{h}$(expr(M_i), l) $\bowtie \mathfrak{h}$(expr(M_i), l)
$\neg$expr(M_i)	$\neg\mathfrak{h}$(expr(M_i), l)
(expr(M_i))	($\mathfrak{h}$(expr(M_i), l))
symm-expr, where e has form local-expr(N)	$\mathfrak{h}$(symm-expr)
constant	constant
name (where name is a global variable)	name
$\sum_{1\le j\le \#N} e_j$	$\sum_{l\in S(N)}(eval(l,e) * \texttt{count\_}N\texttt{\_}f_N(l))$
$\prod_{1\le j\le \#N} e_j$	$\prod_{l\in S(N)}(eval(l,e)$**count_$N$_$f_N(l))$
$\bigwedge_{1\le j\le \#N} e_j$	$\sum_{l\in \mathsf{SAT}_N(e)}$ count_N_$f_N(l) = \#N$
$\bigvee_{1\le j\le \#N} e_j$	$\sum_{l\in \mathsf{SAT}_N(e)}$ count_N_$f_N(l) > 0$
symm-expr $\bowtie$ symm-expr	$\mathfrak{h}$(symm-expr) $\bowtie \mathfrak{h}$(symm-expr)
$\neg$symm-expr	$\neg\mathfrak{h}$(symm-expr)
(symm-expr)	($\mathfrak{h}$(symm-expr))

Figure 6.6: Rules for translating synchronised SPSL statements to a generic form.

where $v^k \in var(M)$. For each state l^j we can compute the local state l'^j reached by executing all variable updates. Then for each state $l \in S(M)$ we compute

$$d_l = \sum_{l=l^i} w^i - \sum_{l=l'^j} w^j$$

which is the difference between the number of modules that were in state l before the transitions and the number of modules that are in l after the transitions. The updates would then be translated as concurrent updates to counter variables of the form

$$\texttt{count\_}M\texttt{\_}f_M(l)\texttt{:=count\_}M\texttt{\_}f_M(l) - d_l.$$

We note that if $d_l = 0$ the change to the counter variable can be omitted.

6.4 Implementation of Synchronisation in GRIP

We have implemented our new translation techniques in a new version of GRIP: GRIP 3.0. It takes a PRISM specification written in a syntax analogous to SPSL as input, which is now allowed to contain any number of synchronised statements, and produces a reduced generic specification which can be used for model checking with PRISM. The synchronised statements do not need to belong to a single synchronised block but can synchronise on different labels.

6.4.1 Implementation

We briefly discuss some techniques and optimisations applied in implementing the translation rules for synchronised statements discussed above. These have no impact on the number of states and transitions of the reduced specification but drastically reduce the amount of time needed for the translation process and the size of the output PRISM specifications.

```
1  dtmc
2
3  module sensor1
4
5    // local state
6    s1 : [0..5] init 0;
7
8    [] s1=0 -> 0.3 : (s1'=1) + 0.3 : (s1'=2) + 0.4 : (s1'=3) ;
9    [a] s1=1 -> (s1'=4);
10   [a] s1=2 -> (s1'=4);
11   [a] s1=3 -> (s1'=5);
12   [] s1=4 -> (s1'=4);
13   [] s1=5 -> (s1'=5);
14
```

```
15 endmodule
16
17
18 module sensor2 = sensor1[s1=s2,s2=s1] endmodule
19 module sensor3 = sensor1[s1=s3,s3=s1] endmodule
```

Listing 6.7: Example specification with overlapping synchronisation statements

First, we consider the possibility that different local states can satisfy the local part of guards of statements with matching updates and symmetric parts of their guard. This can be done explicitly from two different statements with identical updates as shown in Listing 6.7 or could also arise from a single statement involving an assignment based on a global variable, as GRIP considers global variables on a case by case basis in the translation process.

```
1  [a] (no_0=3) -> (no_0'=0) & (no_3'=min(no_3+3,3));
2  [a] (no_0=2) & (no_1=1) -> (no_0'=0) & (no_1'=0) & (no_3'=min(no_3+3,3));
3  [a] (no_0=2) & (no_2=1) -> (no_0'=0) & (no_2'=0) & (no_3'=min(no_3+2,3)) &
     (no_4'=min(no_4+1,3));
4  [a] (no_0=1) & (no_1=2) -> (no_0'=0) & (no_1'=0) & (no_3'=min(no_3+3,3));
5  [a] (no_0=1) & (no_1=1) & (no_2=1) -> (no_0'=0) & (no_1'=0) & (no_2'=0) & (
     no_3'=min(no_3+2,3)) & (no_4'=min(no_4+1,3));
6  [a] (no_0=1) & (no_2=2) -> (no_0'=0) & (no_2'=0) & (no_3'=min(no_3+1,3)) &
     (no_4'=min(no_4+2,3));
7  [a] (no_1=3) -> (no_1'=0) & (no_3'=min(no_3+3,3));
8  [a] (no_1=2) & (no_2=1) -> (no_1'=0) & (no_2'=0) & (no_3'=min(no_3+2,3)) &
     (no_4'=min(no_4+1,3));
9  [a] (no_1=1) & (no_2=2) -> (no_1'=0) & (no_2'=0) & (no_3'=min(no_3+1,3)) &
     (no_4'=min(no_4+2,3));
10 [a] (no_2=3) -> (no_2'=0) & (no_4'=min(no_4+3,3));
```

Listing 6.8: Example reduced specification with overlapping synchronisation statements

Listing 6.8 shows the translation for the block of synchronised statements performed according to the translation rules specified above. Notice that lines $\{1,2,4,7\}$, $\{3,5,8\}$ and $\{6,9\}$ of that listing are groups of statements that perform identical updates. The reason for this is that in this block of synchronised statements the states tracked by counter variables `no_0` and `no_1` have identical behaviour. To reduce this duplication of statements in the reduced specification, we can combine any identically behaving states in guard expressions such as `no_0+no_1=x` where x is the sum of the original assignment values. Listing 6.9 shows the translated synchronised block after this optimisation has been applied.

```
1 [a] (no_0+no_1=3) -> (no_0'=0) & (no_1'=0) & (no_3'=min(no_3+3,3));
2 [a] (no_0+no_1=2) & (no_2=1) -> (no_0'=0) & (no_1'=0) & (no_2'=0) & (no_3'=
    min(no_3+2,3)) & (no_4'=min(no_4+1,3));
3 [a] (no_0+no_1=1) & (no_2=2) -> (no_0'=0) & (no_1'=0) & (no_2'=0) & (no_3'=
    min(no_3+1,3)) & (no_4'=min(no_4+2,3));
```

```
4 [a] (no_2=3) -> (no_2'=0) & (no_4'=min(no_4+3,3));
```

Listing 6.9: Example reduced specification with overlapping synchronisation statements

While the number of statements has reduced from 10 to 4, we note that the underlying size of the model has not changed: this is only a more concise method to specify the same transitions. This optimisation was instrumental in the translation of a specification of an IEEE CSMA/CD communication protocol [147]. The output specification had a 60% reduction in the number of commands and was produced in half the time. More information is given in Section 6.5.3.

The second optimisation performed focused on removing statements whose guards could never be satisfied. Such statements exist because during the translation process, for a given weak composition **w**, we find the corresponding non-local guards, translate them and combine them. However, no checks are performed on those expressions to check whether they can be satisfied: e.g., one statement could have the guard that a global variable *FTR* has value 1, while another requires that global variable to have value 2, so the resulting expression `(FTR=1)` $\wedge$ `(FTR=2)` can never be satisfied.

To combat this, before a translated statement is generated, the reduced fully symmetric expressions are evaluated. If a value of `false` is obtained, then the current statement is discarded and the translation process continues with the next weak composition. Without this functionality PRISM was unable to successfully build a model as it would detect and report statements with guards that cannot be satisfied; however, this optimisation significantly reduced the sizes of the reduced specifications and also reduces the time needed for parsing and building a model based on those specifications.

6.5 Experimental Results

6.5.1 Past examples

First we run GRIP 3.0 on the seven case studies presented on the GRIP webpage. These are model specifications without any synchronised statements which GRIP previously was able to translate correctly. We do this to both show that the updates of GRIP 2.1 have correctly restored GRIP's functionality, and that none of the translation features have had any adverse side effects. Figure 6.7 shows results produced by GRIP 3.0 on those seven models and compares those results with those produced by PRISM and PRISM-symm. The experiments were performed on a 2.60 GHz PC with 16 GB RAM, running PRISM version 4.5 under Windows. The Java maximum memory was set to 6 GB and the maximum memory of the CUDD library set to 1 GB.

Comparing these results to the ones from the introduction of GRIP 2.0 [58], we notice a number of differences. Firstly, the number of states (model sizes) should be identical to those listed in [58] as that is a property inherent to the model and not based on the machine it is run

Case study ($\|S(M)\|$)	#M	Model size (states)		Build time (sec.)			Model check time (sec.)		
		Full model	Symm. reduced	PRISM	PRISM -symm	GRIP	PRISM	PRISM -symm	GRIP
byzantine (10)	6	3.1e+6	31,537	2.009	2.52	4.54	5.438	0.518	0.004
	8	6.4e+8	298,993	7.323	7.32	11.48	2845.27	3.098	2.26
	12	1.0e+13	7.99e+6	OOM	112.53	27.99	OOM	44.562	29.20
	16	OOM	1.08e+8	OOM	OOM	64.16	OOM	OOM	243.21
consensus (6)	6	1.26e+6	12,313	0.104	0.144	0.154	0.005	0.004	0.009
	8	6.10e+7	46,482	0.243	0.384	0.44	0.012	0.011	0.018
	12	1.20e+11	339,729	0.998	1.52	1.45	0.033	0.054	0.057
	16	2.08e+14	1.50e+6	4.21	5.87	4.20	0.089	0.081	0.099
fgf (19)	4	216,961	12,397	0.578	1.34	2.92	24.19	15.02	62.07
	5	4.33e+6	58,411	3.461	5.31	8.68	613.62	220.32	361.86
	6	7.85e+7	212,856	12.06	17.16	14.65	>1h	1104.21	1205.70
	7	1.27e+9	622,262	20.55	32.41	18.92	>1h	3566.57	2239.51
leader (3)	20	3.48e+9	231	0.099	0.208	0.018	0.132	0.029	0.003
	60	4.23e+28	1,891	ODD-x	ODD-x	0.043	ODD-x	ODD-x	0.012
	100	5.15e+47	5,151	ODD-x	ODD-x	0.056	ODD-x	ODD-x	0.028
	140	6.26e+66	10,011	ODD-x	ODD-x	0.098	ODD-x	ODD-x	0.062
mutex (16)	12	4.90e+12	892,542	2.192	2.483	2.728	1.014	0.307	0.423
peer2peer (32)	4	1.05e+6	52,360	0.024	0.12	0.51	2.244	3.212	0.829
	5	3.35e+7	376,992	0.046	0.51	0.634	113.596	25.072	17.42
	6	1.07e+9	2.32e+6	0.067	0.88	1.21	≈12-20h	131.04	130.09
	7	3.44e+10	1.26e+7	0.112	1.61	1.06	>24h	669.98	1087.74
rabin (17)	4	201,828	11,130	0.288	0.77	5.30	0.079	0.091	0.189
	5	6.76e+6	87,312	1.11	2.60	15.04	0.23	0.24	0.43
	6	1.30e+8	356,592	1.99	5.14	21.79	0.41	0.70	0.96
	8	4.51e+10	4.06e+6	5.69	13.22	44.59	1.09	0.81	3.167

Figure 6.7: Comparison of experimental results for GRIP 3.0, PRISM and PRISM-symm. Cells labelled with OOM represent models for which PRISM reported an out-of-memory error. Models for which PRISM could not construct the underlying Ordered Decision Diagram (ODD) due to running out of memory are labelled as ODD-x.

on or the tool that is used. This is true in most cases both with respect to the full model and the symmetry reduced model. Differences arise in the five, six and seven symmetric module instances of the *fgf* model. As the GRIP webpage [97] and the GRIP source code base [96] only provide one example specification per case study, we needed to re-generate the specifications for a variety of numbers of symmetric modules. This was trivial in some cases, where it only required the addition of new module renaming commands; however, in some cases the behaviour of a symmetric module depends on the number of modules present. This could, for example, result in different variable ranges for some or all variables of the model. As GRIP does not support the declaration of constants and labels, these relationships have become obfuscated in the specifications. We have examined the relevant case studies on the PRISM webpage [193] to find the missing declarations and discover those relationships, such as the number of data blocks to be downloaded by the *peer2peer* protocol and the shared counter variable of the *rabin* model, which depends on the logarithm of the number of symmetric modules, so was causing issues only in the four module instance. However, the case study related to the *fgf* model [109] gives a completely different specification from the fully symmetric one used by GRIP. We choose to keep *fgf* as a case study as we believe the differences to result in a change in the verification results and not prevent us from discussing the space and time performance of the models. Lastly, the 16 module instance of the *byzantine* model now results in an out-of-memory error before computing the number of states of the model. We have been unable to figure out why this is the case, but will continue to investigate.

Next, we compare the build times and model checking times. The models for the *rabin*, *peer2peer* and *fgf* all have results similar to the original experiments, with differences explained by incremental improvements of PRISM and the hardware used over the years since the original experiments. The increase in the times for the *fgf* model are likely due to the incorrect parameters mentioned above.

The *leader* case study presents an unexpected change. For instance sizes of 60, 100, and 140 symmetric modules, PRISM now reports the following error message:

```
Cannot construct ODD for this model, number of states too large
```

This happens both for the full model checking with PRISM and the symmetry reduced with PRISM-symm, despite both previously working correctly. *leader* is a very simple and highly symmetric model with only three local states, which means that it can be used to model large numbers of symmetric modules. We note that the full model size is much larger than that of the other case studies. It is likely that a new feature has been added to PRISM that prevents the compilation of models for which the number of states is too large. Neither the GRIP webpage, nor the paper related to it [65] mention any PRISM runtime flag that should be set to avoid this issue. As PRISM-symm requires a symbolic representation for the full model to be built before reducing it to a quotient model, it encounters the same issue during the initial construction of

Case study ($\|S(M)\|$)	#M	Model storage (MTBDD nodes)			Verification result		
		PRISM	PRISM -symm	GRIP	PRISM	PRISM -symm	GRIP
byzantine (10)	6	158,044	49,428	32,138	0.5	0.5	0.5
	8	732,727	164,218	87,716	0.5	0.5	0.5
	12	OOM	859,184	427,407	n/a	0.0	0.0
	16	OOM	OOM	1.04e+6	n/a	n/a	0.0
consensus (6)	6	6,909	7,083	7,235	0.0	0.0	0.0
	8	15,529	15,883	16,657	0.0	0.0	0.0
	12	50,037	50,741	39,454	0.0	0.0	0.0
	16	115,385	116,691	86,570	0.0	0.0	0.0
fgf (19)	4	56,579	189,815	343,656	4.81e-7	4.81e-7	4.81e-7
	5	112,038	509,516	654,848	4.83e-7	4.83e-7	4.83e-7
	6	174,979	1.04e+6	1.03e+6	n/a	4.84e-7	4.84e-7
	7	240,720	1.69e+6	1.44e+6	n/a	4.85e-7	4.85e-7
leader (3)	20	5,300	3,789	626	1.0	1.0	1.0
	60	ODD-x	ODD-x	1,602	ODD-x	ODD-x	1.0
	100	ODD-x	ODD-x	2,858	ODD-x	ODD-x	1.0
	140	ODD-x	ODD-x	4,618	ODD-x	ODD-x	1.0
mutex (16)	12	40,687	23,459	32,211	true	true	true
peer2peer (32)	4	11,931	84,932	84,272	0.175...	0.175...	0.175...
	5	26,246	197,306	157,468	0.215...	0.215...	0.215...
	6	40,561	359,487	247,114	n/a	0.232...	0.232...
	7	56,142	580,889	355,713	n/a	0.235...	0.235...
rabin (17)	4	65,624	96,559	133,531	true	true	true
	5	136,840	257,446	355,240	true	true	true
	6	206,213	408,291	554,360	true	true	true
	8	381,184	796,324	1.34e+6	true	true	true

Figure 6.8: Model storage and verification results for GRIP 3.0, PRISM and PRISM-symm.

the full model. Since GRIP only pre-processes model specifications, PRISM manages to run successfully on the smaller symmetry reduced model (of up to 10,000 states).

In the case of the *consensus* case study there is a large discrepancy between the current results and those presented in [58]. In all cases property verification is completed in a fraction of a second, whereas the full model without symmetry reduction was listed as taking more than a day. Investigating the Case Studies webpage of the PRISM website, we find that the *consensus* specification is based on an earlier case study by Kwiatkowska et al. [145]. We notice that this case study actually employs synchronisation for a single transition command:

```
[done] (pc1=3) -> (pc1'=3);
```

Furthermore, the verification results reported for the property (see Fig. 6.9) are varied and are very different from the 0.0 values obtained by us (see Fig. 6.8). A lower bound for the value of this property is established in [10], so a zero value is proven to be incorrect. It is possible that the synchronisation label was mistakenly removed when *consensus* was originally selected as a case study for GRIP 2.0 which did not support synchronisation. Interestingly, we find that the *consensus* model is also one of the four case studies presented by PRISM-

Model:		p:	(K-1)/2K:
N	K		
2	2	0.382814	0.25
	4	0.437752	0.3125
	8	0.468783	0.4375
	16	0.484507	0.46875
	32	0.492715	0.484375
	64	0.498193	0.492188
4	2	0.317360	0.25
	4	0.406307	0.3125
	8	0.453255	0.4375
	16	0.477088	0.46875
	32	0.490379	0.484375
6	2	0.294365	0.25
	4	0.395906	0.3125
	8	0.448209	0.4375
	16	0.475138	0.46875
8	2	0.282790	0.25
	4	0.390749	0.3125
	8	0.445831	0.4375
	16	0.474747	0.46875
10	2	0.275919	0.25
	4	0.387693	0.3125
	6	0.425450	0.416667

Figure 6.9: Verification results from the *consensus* case study [145].

Image taken from https://www.prismmodelchecker.org/casestudies/consensus_prism.php

symm. Surprisingly, despite PRISM-symm supporting synchronisation, all five instances of the model are also missing the synchronisation label, and consequently obtain the wrong verification results.

Making use of the ability of GRIP 3.0 to apply symmetry reduction to specifications with synchronisation labels, we reran the instances from [145]. Instances are specified by N, the number of symmetric modules, and a constant parameter K, which is used to determine the boundary conditions for the counter used by the protocol. Figure 6.10 shows the model sizes and the times taken for property verification. The verification results are within 0.01 of those originally presented (see again Fig. 6.9) in all cases (larger models have larger deviations).

6.5.2 Rock-Paper-Scissors

We consider a model of a Rock-Paper-Scissors game. Modules represent participants, who choose between three options: rock, paper and scissors. When all choices have been made, they are evaluated. If all choices are different or all are the same, the game continues for another round. Otherwise an outcome is announced, based on the choices made. Synchronised statements are required to ensure that all choices are made before the result is evaluated. The PRISM

Consensus (N,K)	Model size (states)		Model check time (sec.)			Verification result (if appl.)
	PRISM	Reduced model	PRISM	PRISM -symm	GRIP	
2, 2	272	154	0.01	0.005	0.005	0.382
2, 4	528	298	0.026	0.011	0.013	0.437
2, 8	1,040	586	0.09	0.045	0.043	0.468
2, 16	2,064	1,162	0.42	0.279	0.262	0.484
2, 32	4,112	2,314	2.69	1.65	1.46	0.491
2, 64	8,208	4,618	17.63	10.21	9.35	0.493
4, 2	22,656	2,151	0.71	0.118	0.082	0.317
4, 4	43,136	4,087	4.15	0.493	0.381	0.406
4, 8	84,096	7,959	29.28	2.83	2.40	0.452
4, 16	153,216	14,493	154.75	15.58	12.64	0.474
4, 32	329,856	31,191	1,940	123.7	99.32	0.486
6, 2	1.25e+6	12,313	145.9	1.27	0.715	0.294
6, 4	2.38e+6	23,233	1,054	7.23	4.25	0.395
6, 8	4.61e+6	45,073	12,220	49.12	27.4	0.447
6, 16	9.09e+6	88,753	>24h	313.2	176.9	0.472
8, 2	6.10e+7	46,482	19,239	8.68	4.16	0.282
8, 4	1.15e+9	87,378	>24h	55.73	26.74	0.390
8, 8	2.22e+9	169,170	>24h	379.9	175.1	0.444
8, 16	4.37e+9	332,754	>24h	3,735	1,567	0.470
10, 2	2.76e+9	136,708	>24h	45.52	19.56	0.275
10, 4	5.18e+9	256,388	>24h	349.2	158.8	0.387
10, 8	1.00e+10	495,748	>24h	3,159	1,201	0.442

Figure 6.10: Model size and verification results for the Randomised Consensus Shared Coin Protocol obtained by GRIP 3.0, PRISM and PRISM-symm.

model is shown in Listing 6.10.

```
dtmc
global r : [0..1] init 0;
global p : [0..1] init 0;
global s : [0..1] init 0;

module player1
// choice: 0-undecided, 1-rock, 2-paper, 3-scissors
ch1 : [0..3];
// local phase
ph1 : [1..2];
// winner: 1-rock, 2-paper, 3-scissors
res1 : [0..3];
// make choice
[](ch1=0)&(ph1=1)&(res1=0)) -> 1/3: (ch1'=1) & (r'=1)
                             + 1/3: (ch1'=2) & (p'=1)
                             + 1/3: (ch1'=3) & (s'=1);
// determine outcome
[decided] ((ph1=1) & (res1=0)) -> (ph1'=2) ;
[](ph1=2)&(res1=0))&((r=1)&(p=0)&(s=1))->(ch1'=0)&(res1'=1);
[](ph1=2)&(res1=0))&((r=1)&(p=1)&(s=0))->(ch1'=0)&(res1'=2);
[](ph1=2)&(res1=0))&((r=0)&(p=1)&(s=1))->(ch1'=0)&(res1'=3);
```

RPS	Model size (MTBDD)			Model build time (sec.)			Model check time (sec.)		
	PRISM	PRISM	GRIP	PRISM	PRISM	GRIP	PRISM	PRISM	GRIP
m		-symm			-symm			-symm	
2	453	280	605	0.03	0.066	0.01	0.01	0.03	0.03
3	1774	749	1029	0.04	0.109	0.15	0.01	0.05	0.03
4	4311	1513	2156	0.05	0.112	0.35	0.02	0.04	0.04
5	8021	2394	2880	0.07	0.185	0.88	0.05	0.04	0.05
6	12902	3335	3672	0.08	0.207	2.98	0.15	0.06	0.13
7	18951	4360	4593	0.09	0.339	6.66	0.59	0.07	0.13
8	26153	5442	7240	0.15	0.428	20.73	5.16	0.08	0.08
9	34526	6593	8611	0.16	0.545	30.68	148.21	0.09	0.33
10	44067	7816	10198	0.18	0.712	54.09	261.71	0.12	0.55

Table 6.3: Model size and build times for the Rock-Paper-Scissors model for m participants, obtained by PRISM, PRISM-symm and GRIP 3.0.

```
22 [](ph1=2)&(res1=0))&((r=0)&(p=0)&(s=0))->(ch1'=0);
23 [](ph1=2)&(res1=0))&((r=1)&(p=0)&(s=0))->(ch1'=0)&(r'=0)&(p'=0)&(s'=0);
24 [](ph1=2)&(res1=0))&((r=0)&(p=1)&(s=0))->(ch1'=0)&(r'=0)&(p'=0)&(s'=0);
25 [](ph1=2)&(res1=0))&((r=0)&(p=0)&(s=1))->(ch1'=0)&(r'=0)&(p'=0)&(s'=0);
26 [](ph1=2)&(res1=0))&((r=1)&(p=1)&(s=1))->(ch1'=0)&(r'=0)&(p'=0)&(s'=0);
27 // reset for next round if needed
28 [reset] ((ch1=0)&(ph1=2)&(res1=0)) -> (ph1'=1) ;
29 endmodule
30 module player2=player1[ch1=ch2,ch2=ch1,ph1=ph2,ph2=ph1,res1=res2,res2=res1]
    endmodule
```

Listing 6.10: Rock-Paper-Scissors model. Multiple module renamings are not shown.

Table 6.3 shows the model sizes and execution times for the Rock-Paper-Scissors model described above for m participants, for $2 \leq m \leq 10$, using PRISM, PRISM-symm and GRIP respectively. The property verified is: "what is the probability that the winning outcome is *rock?*".

Compared to PRISM, the GRIP specification is more complex in all cases but the resulting model has far fewer states (when $m > 2$). Consequently the build times increase for reduced models and the times taken for model checking decrease. On this example GRIP is significantly out-performed by PRISM-symm. We suspect that this is because, given the size of the example, the synchronisation dominates. (Recall from Lemma 2 that synchronisation results in an exponential increase in translated statements). We expect our approach to be most beneficial for models that involve a majority of non-synchronised statements. However, we do achieve a significant improvement in comparison to standalone PRISM. The example also serves to demonstrate the correctness of GRIP's new support for synchronisation.

6.5.3 PRISM-symm case studies

Previously, the wider applicability of PRISM-symm has meant that it has been used more widely. In particular, examples which use synchronisation labels have been out of scope for GRIP. As an example, PRISM-symm has been used to analyse the IEEE 802.3-2002 CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) communication protocol (*csma*) [120] . It is a communication protocol designed to reduce data collisions (simultaneous uses of the channel) in networks with a single channel by determining the behaviour of devices participating in the protocol. The basic operation of the protocol is as follows: when a device wants to send data, it first listens to the medium. If no one is transmitting on the medium, the device starts to send its data. On the other hand, if the medium was sensed busy, the device backs off for a random amount of time and then repeats this process. The same communication protocol has been used previously as a case study for symbolic model checking using PRISM [148]. Further details are available from PRISM's webpage [193]. We attempt to apply GRIP 3.0 to that example in order to learn the extent of the effect of the new features.

Examining the *csma* case study we notice that it is the type of model that GRIP is not well suited for. GRIP excels at a larger number of symmetric copies of a simpler module, while PRISM-symm is best at a smaller number of more complex modules. The *csma* specification is of the latter type. The symmetric module has $|S(M)| = 118$ local states, an order of magnitude larger than the typical GRIP case studies. Additionally, most of its synchronised statements do not have a highly restrictive guard; for example, a single synchronised statement can have its guard satisfied by modules in 60 out of the 118 possible states. This causes an exponential increase in the number of weak compositions that need to be considered for statements with that synchronisation label (see Equation 6.1).

Our first attempt to apply GRIP 3.0 to this example resulted in approximately eight million weak compositions being generated for a single synchronisation label of the *csma* specification. This process would require more than an hour to complete and result in the same number of reduced transition statements in the reduced specification. The two optimisations described in Section 6.4.1 allowed a specification to be successfully generated in a smaller amount of time; however, as previously mentioned, the number of resulting transitions remains unchanged.

We conclude that while theoretically our approach could be applied to the *csma* case study, it is better suited to models with less complex symmetric modules. Additionally, the guards of synchronised statements are preferred to be as restrictive as possible. As a result, our improvement to GRIP makes the tool more widely applicable in the area of models it most excels at, namely, models with a large number of simpler modules which communicate through a simple synchronised mechanism.

Byz	Model size (MTBDD)			Model build time (sec.)			Model check time (sec.)		
	PRISM	PRISM	GRIP	PRISM	PRISM	GRIP	PRISM	PRISM	GRIP
m		-symm			-symm			-symm	
6	130,145	54,512	21,018	1.36	0.207	1.29	2.07	0.30	0.26
8	592,630	214,293	54,887	6.20	0.428	1.92	>10m	1.39	0.66
12	OOM	OOM	218,153	OOM	OOM	2.98	OOM	OOM	4.653
16	OOM	OOM	343,941	OOM	OOM	6.06	OOM	OOM	13.84

Table 6.4: Model size and build times for the Byzantine model obtained by PRISM, PRISM-symm and GRIP 3.0. OOM signifies models which resulted in an Out-of-Memory error.

6.5.4 Randomised Byzantine Agreement protocol

Our final example before applying GRIP to the Ctrl-MAC protocol is an adaptation of the Byzantine agreement protocol [33]. The original protocol was an example for which GRIP performed favourably compared to PRISM-symm [59]. We have added a common synchronisation label to three of the statements that have updates to local states only and compare performance again. We do so to demonstrate GRIP's performance on a real-world scenario. Results for a range of numbers of participants *m* are shown in Table 6.4. Despite the additional synchronisation, GRIP still performs well for this example compared with PRISM-symm.

6.5.5 Ctrl-MAC models

We now apply GRIP 3.0 to the Ctrl-MAC models described in Section 5.1 which motivated our research into GRIP. We use model 3, our best performing model without the use of counter abstraction, as input to GRIP. We begin with a basic Ctrl-MAC configuration involving 5 request slots and 5 sensor nodes.

During this process we encountered a number of issues that we had to resolve before we were able to run GRIP on the Ctrl-MAC models. All of the current example models for GRIP (see Table 6.2) do not use any `const` or `formula` declarations and in fact GRIP 2.0 does not support their use. In [65] the integration of this part of the PRISM language is left as future work and the example models used are manually refactored to remove the uses of these structures. As our Ctrl-MAC models use formulas to a large extent, we introduce a preprocessing script to GRIP 3.0 that refactors any input PRISM model such that all occurrences of `const` or `formula` are replaced with their values. We do not believe that introducing support for these two structures to GRIP would provide any significant performance improvements. If that were the case, GRIP would need to substitute any constants and calculate every formula each time it encounters them in a transition command in order to find the counters corresponding to the two states involved in that transition.

Another issue is that GRIP does not support the use of in-line if statements, whereas our Ctrl-MAC models heavily rely on them in order to calculate the number request slots that have experienced a conflict and the back-off duration that should be applied to each sensor node. For

example, the number of conflicts in an RRM is given by

```
formula new_contentions = ((c0=2?1:0) + (c1=2?1:0)
    + (c2=2?1:0) + (c3=2?1:0) + (c4=2?1:0));
```

We have two possible solutions to this issue. The first is to replace each transition command involving an in-line if statement with two transition commands: one for the true case and one for the false case. For example, the command

```
[] s1 = 6 -> (s1'=((backoff=0)?7:2));
```

can be substituted by the following two commands

```
[] s1 = 6 & backoff=0 -> (s1'=7);
[] s1 = 6 & backoff>0 -> (s1'=2);
```

where each has an additional expression in their guard and a corresponding change to the resulting update. This option can be implemented as a preprocessing script, and while this is the simpler solution, it also can result in an exponential increase in the number of transition statements used by the model. This change would typically have no effect on the size of the model, but does heavily influence the size of GRIP 3.0's output model. Specifically for the Ctrl-MAC models, the in-line if statements are mainly used when sensors are assigned back-off durations, which are all synchronised commands as back-offs are assigned simultaneously as the RRM is transmitted.

Sensors	GRIP Time (s)	Weak Compositions	Output model size	Output file size
2	1.1	7326	OOM	7515
3	14.2	283,765	OOM	283,964
4	OOM	≈8e+6	n/a	n/a
5	OOM	≈8e+8	n/a	n/a

Table 6.5: Model sizes and computation times of GRIP to preprocessed models of Ctrl-MAC. Cells labelled with OOM represent models for which PRISM or GRIP reported an out-of-memory error.

We use this approach to investigate its performance on the Ctrl-MAC models. Table 6.5 shows the results obtained by GRIP. The number of weak compositions reported are the maximum possible as calculated by Equation (6.1). It is likely that many of these will not result in valid transitions. Models with four or more sensor modules resulted in GRIP running out of memory during the translation process due to an exponential increase in the number of weak compositions considered. Models involving only two and three sensor nodes were successfully generated; however, they were unable to be built by PRISM as they resulted in out-of-memory errors. The input model used is listed in Appendix A.1.

Subsequently, we reduce the number of request slots to two - the smallest possible number of request slots that exhibits Ctrl-MAC's back-off strategy. As mentioned in Chapter 5 existing wireless medium regulations require Ctrl-MAC implementations to use at least five request slots, so such a configuration is unusable in practice; however, it can serve as a proof of concept for our approach. Table 6.6 shows verification results for Ctrl-MAC models ranging from two to five sensors.

As stated previously, our Ctrl-MAC models treat the environment in two ways: either no new data is detected until all conflicts are resolved, or sensors detect new data upon successful transmission. We present the verification results from two fully symmetric properties: 1 "what is the probability that all sensor nodes simultaneously choose a given request slot", and 2 "what is the probability that all sensor nodes experience congestion during an RRC". The models produced by GRIP achieve the same verification results, while obtaining a drastic decrease in the number of states and transitions. In fact, we note that the sizes of the state spaces of the models produced by GRIP are linearly related. However, despite this decrease, the symmetry-reduced models reach an out-of-memory error when built by PRISM much sooner than their non-reduced counterparts. Despite the lower number of states and transitions, the number of nodes in the transition matrix of the models is often higher than that of the original model.

Reducing the number of request slots to two allowed us to investigate a larger suite of models; however, even in this case we were unable to verify a protocol configuration of a large number of sensor nodes. Even though our Ctrl-MAC models were expected to be well suited for counter abstraction performed by GRIP, i.e. they model a large number of devices that are simple in their control logic, their synchronised blocks are not. We introduced synchronisation to GRIP in a way contrary to the usual counter abstraction methodology. Synchronised blocks need to explicitly bridge states occupied by modules prior to the execution of the block to those occupied after execution. As such, models translated by GRIP should contain synchronised blocks that are as simple as possible. This is not the case for our Ctrl-MAC models. When a Request-Reply Message occurs, a sensor node can have a successful request, an unsuccessful request, or be backed off from a previous conflict. Even with only two request slots, each of these scenarios admits a variety of local and global states according to: FTR value, number of requests per request slot, back-off to be assigned. This results in an increase in size of the transition matrix of the symmetry-reduced model. The current method of reducing synchronisation with GRIP allows us to handle models that have a small number of synchronised commands as a secondary aspect of the model. However, it is not well suited to our Ctrl-MAC models for which a large part of the complexity lies in synchronised statements.

Sensors (version)	Time (s)	States	Transitions	Matrix (nodes)	PCTL result
The models below do not generate new events					
2 (normal-1)	0.059	47	81	564	0.333
2 (grip-1)	1.45	29	44	2220	0.333
2 (normal-2)	0.059	47	81	564	0.5
2 (grip-2)	1.45	29	44	2220	0.5
3 (normal-1)	0.044	449	979	1855	0.285
3 (grip-1)	12.025	113	197	5497	0.285
3 (normal-2)	0.044	449	979	1855	0.249
3 (grip-2)	12.025	113	197	5497	0.25
4 (normal-1)	0.077	4047	10389	6155	0.179
4 (grip-1)	n/a	OOM	OOM	n/a	n/a
4 (normal-2)	0.077	4047	10389	6155	0.5
4 (grip-2)	n/a	OOM	OOM	n/a	n/a
The models below generate events continuously					
2 (normal-1)	0.031	55	99	672	0.333
2 (grip-1)	0.49	33	53	2739	0.333
2 (normal-2)	0.031	55	99	672	0.5
2 (grip-2)	0.49	33	53	2739	0.5
3 (normal-1)	0.036	215	547	1658	0.143
3 (grip-1)	3.23	63	119	5225	0.143
3 (normal-2)	0.036	215	547	1658	0.249
3 (grip-2)	3.23	63	119	5225	0.25
4 (normal-1)	0.048	881	2424	5840	0.066
4 (grip-1)	5.171	106	193	5946	0.066
4 (normal-2)	0.048	881	2424	5840	0.5
4 (grip-2)	5.171	106	193	5946	0.5
5 (normal-1)	0.123	10671	35447	13,737	0.032
5 (grip-1)	18.688	208	411	9772	0.032
5 (normal-2)	0.123	10671	35447	13,737	0.687
5 (grip-2)	18.688	208	411	9772	0.687
6 (normal-1)	0.141	32335	119071	33,626	0.015
6 (grip-1)	n/a	OOM	OOM	n/a	n/a
6 (normal-2)	0.141	32335	119071	33,626	0.812
6 (grip-2)	n/a	OOM	OOM	n/a	n/a

Table 6.6: Model sizes and computation times for PRISM models of Ctrl-MAC compared to their symmetry reduced counterparts produced by GRIP. The table shows verification results of two properties (-1 and -2 respectively). Cells labelled with OOM represent models for which PRISM reported an out-of-memory error.

6.6 Summary

In this chapter we introduced the GRIP symmetry reduction tool and the improvements we have made. We restored all of the functionality that it was missing and added support for the translation of synchronised commands in order to increase the range of models it can be applied

to. Additionally, we corrected a long-standing issue that prevented the use of division in any expression in models input into GRIP. These updates resulted in two new versions: GRIP 2.1 and GRIP 3.0. We have defined new translation rules for SPSL, the language GRIP is based on, and shown that those are sound in the translation of synchronised statements. We discussed the limitations of the translation method and calculated the order of the size of the translated specification: the output model is not guaranteed to achieve a state space reduction from applying counter abstraction to its synchronised statements. The goal is to allow the process to be applied to models that use synchronisation, so that we can benefit from the state space reduction gained from applying counter abstraction to the non-synchronised transition statements. We expect this to be most beneficial for models that involve a majority of non-synchronised commands and a few synchronised ones.

We measure the effectiveness of our approach by applying GRIP 3.0 to a suite of specifications involving symmetry. We show that we have restored GRIP's functionality on all of the example specifications used for GRIP 2.0. We present results from GRIP 3.0's application to models involving synchronised statements. While our main reason for looking into GRIP, the Ctrl-MAC models presented in Chapter 5, were not well suited for this symmetry reduction process due to an extensive use of synchronised statements, we achieved a reduction in model sizes of some of the specifications. In particular, the *consensus* case study, which was previously incorrectly implemented by both GRIP and PRISM-symm, achieved significant reductions in state space using GRIP 3.0. Its specification used only a single synchronised statement, making it an ideal application for our translation method.

6.6.1 GRIP future work

We briefly outline some ideas for future versions of GRIP. First, even though SPSL translation rules are designed so that any symmetric PRISM model can be translated to a symmetry-reduced one that has had counter abstraction applied, in practice there are a number of issues that prevent this from happening. These are due to features of PRISM that GRIP does not support and result in either an inability of GRIP's parser to parse the model or a failure of some of GRIP's safety checks. We present a brief list of some of the issues that we have identified:

- PRISM's variables can be boolean. GRIP's grammar supports boolean expressions, but those expressions cannot be assigned to a variable as the syntax for declaring a boolean variable is not supported.
- PRISM allows negative values to be assigned to its variables whereas GRIP does not.
- PRISM's module renaming rules are less strict that those of GRIP. GRIP requires module renamings declarations to be of the form "`module M2 = M1 [ x=y, y=x ] endmodule`" which is the format listed on the PRISM website. However, investigating

the case studies on the PRISM website, we notice that renamings are typically written in only one direction, i.e. "`module M2 = M1 [ x=y ] endmodule`". This is likely due to a change in PRISM that has been introduced since the release of GRIP 2.0.

- The names of global variables cannot contain digits. While this is not an error that limits the range of models GRIP can be applied to, we believe that this change could make the use of the tool more convenient.

- PRISM allows a PRISM model to start with a comment rather than a model type declaration. GRIP's grammar correctly identifies comments as tokens that should be ignored; however, there is an issue that prevents this from happening prior to the model type declaration, which the algorithm expects to be the first line of the input specification.

While GRIP currently does not support translation of steady-state properties, it can be easily extended to do so. We have manually translated and verified steady-state properties for some of the models described above as a proof of concept, and we wish to formally describe and implement this feature into GRIP. Similarly, we could introduce support for analysis of properties based on costs and rewards into GRIP. Any rewards structure would need to be well-defined in order to be translated by GRIP (i.e. any expressions used cannot reference individual symmetric modules, similar to rules in Figure 3.5). Additionally, GRIP and SPSL would need to be extended to support the translation of reward-based properties. This would increase the expressiveness of properties that can be verified by GRIP.

One of the main building blocks for our approach to synchronised statements in GRIP 3.0 is the generation of weak compositions of states. At present, our algorithm goes through all possible weak compositions and discards those which would result in commands whose guards can never be satisfied. This is caused by conflicts in the global part of the guards of two or more synchronised statements. For example, consider a symmetric module whose synchronised commands are based on a global variable `switch`

```
[action] s1=1 & switch=0 -> (s1'=2);
[action] s1=1 & switch=1 -> (s1'=3);
```

These two commands are mutually exclusive as when this synchronised transition is executed, it is impossible for symmetric modules to satisfy the guards of both statements. Cases such as this one occur more frequently than expected due to the way in which GRIP translates commands which use global variables in the expressions for the updates of local variables. GRIP uses *valuations*, so the command

```
[action] (s1=1) -> (s1'=2) & (choice1'=counter+2);
```

where `s1` and `choice1` are local variables, and `counter` is a global variable with a range

from 0 to 3, is considered on a case by case basis for each value of `counter`. Therefore, during the translation process of GRIP, the command above is equivalent to

```
[action] (s1=1) & (counter=0) -> (s1'=2) & (choice1'=2);
[action] (s1=1) & (counter=1) -> (s1'=2) & (choice1'=3);
[action] (s1=1) & (counter=2) -> (s1'=2) & (choice1'=4);
[action] (s1=1) & (counter=3) -> (s1'=2) & (choice1'=5);
```

If GRIP generates weak compositions using a case by case approach based on the set of values of the global variables present in the guards of synchronised statements, it can significantly decrease the worst case complexity of the translation algorithm as the number of weak compositions is exponential in the number of states accepted by the guards of synchronised commands.

Lastly, GRIP's expressions are restricted in comparison to those of PRISM. GRIP's grammar does not support the use of the if-and-only-if (<=>) and condition evaluation (`condition ? a : b`) operators, and only two of the eight PRISM built-in functions (e.g. `min`, `min`, `max`, `floor`, `pow`, `log`, etc). The lack of these features reduces the number of model specifications that can be translated by GRIP without requiring pre-processing by the user.

Chapter 7

Conclusions

In this thesis we investigate the use of formal methods to provide guarantees of correctness for communication protocols used by sensor networks. We find that formal methods are effective for wireless sensor systems of moderate size, and can be used as a basis for the development of communication protocols and can aid in the construction of mathematical results for larger systems. While our research is motivated by its application to sensor systems, the results we produced can be applied to a wider range of systems. We used the Ctrl-MAC sensor network communications protocol as a starting point for our research and the discoveries we arrived at were based on challenges encountered throughout the verification process.

We started our investigation by providing an overview of the current state of CPSs with a focus on WSNs. We looked at the characteristics of a wireless sensor network, and the ways in which networks are classified according to their structure. We briefly discussed where Ctrl-MAC fits in this background, the challenges it was designed to overcome, and the design choices made to achieve this goal. We then presented formal verification technologies and compared available model checking tools. We introduced the state space explosion problem and the techniques most frequently used to overcome it (e.g. partial order reduction, symmetry reduction, etc.). We also reviewed some of the tools that can be used to carry out these techniques. We then gave on overview of how formal verification has been used for the analysis of sensor systems and discussed the difficulties presented by this type of systems.

In chapter 3 we presented all of the definitions and mathematical concepts used in this thesis. We introduced DTMCs and MDPs, the core structures that model checking is based on, and gave a brief overview of the probabilistic logic PCTL, one of the most common ways of expressing desired properties of probabilistic systems, as well as of cost and rewards structures. We introduced the PRISM model checker, the model checker we predominantly use throughout the thesis, and looked in more detail at the syntax of the PRISM modelling language. We then introduced the most frequently used state space reduction techniques and the mathematical concepts they are based on. We presented in more detail two symmetry reduction tools that implement these concepts, PRISM-symm and GRIP. We then focus on GRIP for a large part of

the thesis. We then introduced the Ctrl-MAC protocol in greater technical detail in Chapter 4, and introduced some of the more intricate details that presented a challenge in modelling the protocol.

During the process of creating a model for Ctrl-MAC, a number of conversations with the protocol developers from Imperial College London were carried out, and we were successful in identifying a number of mistakes in the original protocol specification. This was a very important result as the specification would otherwise lead to wrong implementations of the protocol and sub-optimal performance of those implementations, which is a frequent problem in computer networks. We created a suite of PRISM models that could be used to analyse the Ctrl-MAC protocol. These models were incrementally constructed, with each of them introducing improvements to issues faced by the previous iteration. We crafted the individual models so that they would be best suited to model the type of properties that we were interested in, and were subsequently able to successfully model Ctrl-MAC implementations of a moderate number of sensor devices. However, this was still insufficient to analyse the large number of devices that were expected for a typical Ctrl-MAC implementation. We then used the results obtained in the models, and the behavioural patterns observed, in order to devise a statistical approach capable of producing probability distributions for protocols involving a large number of sensors.

We identified GRIP as the symmetry reduction tool that is best suited to perform symmetry reduction on the Ctrl-MAC models. In order to successfully apply GRIP, we needed to first update and extend its functionality. The tool was in a state that prevented it from successfully being used to perform symmetry reduction on any non-trivial PRISM model. We introduced a number of fixes to the tool, and restored it to its previous functionality. Additionally, we extended the tool by providing support for the application of symmetry reduction to models that used synchronised statements. To do so, we developed new language translation rules for the SPSL language and extended its expressiveness. When this new version of GRIP was applied to the models of the Ctrl-MAC protocol, we discovered that Ctrl-MAC was a protocol that was badly suited for the translation process of synchronised statements, due to the high number of states sensors can be in when synchronising during an RRM. Despite not being to apply GRIP to our original goal, our contribution performed well on models that were better suited to the approach we used. This allowed us to find and resolve a long-standing error due to synchronised commands, that was present in a model used as an example specification by both GRIP and PRISM-symm.

7.1 Future work

We identify some avenues for future work, some of which have been previously discussed at greater length in the thesis.

In section 5.3.2 we proposed the use of a population based approach when modelling an

environment which triggers the sensors at an uniform rate. This approach was deemed out of scope for this thesis, but we make note of it here as future work. In addition, we propose the investigation of statistical model checkers for modelling Ctrl-MAC. The properties we are interested in are not rare events, we are in fact interested in the most common behaviour of the model, so the simulation-based approach should be well suited to this problem.

Following the discussion in Section 5.3.3, we established that for an instance of the Ctrl-MAC protocol in which sensors detect new phenomena at a uniform rate, at the equilibrium position the number of successful requests at each RRC is much less than the number of request slots available. Currently, the default Ctrl-MAC implementation consists of five requests slots on one channel, and three data channels each with three slots per RRC. In [22], simulations are performed to show that the use of a dedicated channel for requests is preferred to the free use of all available channels. While in this thesis we have not focused on the transmission of data following a successful transmission request, our results hint that a large number of the data slots available would go unused. Essentially, using a single channel for requests does not enable the full utilisation of the data channels. This is not necessarily a drawback of the protocol, but an investigation could be performed into dedicating two channels for requests. This will not be a straightforward task as currently the requests channel has a 10% duty cycle, while the data channels use a 1% duty cycle, so care must be taken when establishing such a structure. The simulations in [22] showed that using all four channels for both data and requests achieved higher transmission delays and lower packet delivery ratios than having one dedicated channel for requests and three for data transmission. It was also claimed that the dedicated approach allowed for data load to be distributed evenly. We find that even under optimal transmission request conditions, we do not expect enough successful requests to fully utilise the data channels.

Lastly, we have noticed that our models contain a large amount of states that have an extremely small probability of being reached. For example, the probability of the event in which all sensors send transmission requests on the same request slot decreases exponentially in the number of transmitting sensor devices. Furthermore, we have seen that depending on the application of the protocol, it is likely that a protocol never reaches any extremes in its behaviour: i.e. never has all sensor devices backed off, never has all sensor devices transmitting requests at the same time, etc. In [87], an “abstracting the abstraction” approach is proposed for this scenario. This approach is based on the idea that states with small probabilities could be grouped together. For example, instead of keeping track of the exact number of sensor devices that are currently idle for a model with 100 devices, we could instead use a number of ranges we define such as: “less than 10”, “between 10 and 20”, “21”, “22”, “23”, “24”, “between 25 and 50”, “more than 50”. The ranges would be defined such that the total probabilities of each range is similar.

In Section 6.6.1 we outlined a variety of issues encountered by GRIP in the translation of PRISM models and presented possible features that could be added to GRIP to broaden the range of PRISM specifications it can be applied to without the need of any pre-processing. Ad-

ditionally, we proposed GRIP to be extended with support for steady-state and reward-based properties. Such properties are often used in the verification of PRISM models and their translation could be based on the current translation rules for PRISM properties. We also identified that our current implementation of the generation of weak compositions of states can be significantly improved. Due to the nature of the translation process a large number of weak compositions result in guards that can never be satisfied. An optimised algorithm could identify incompatible states and exclude any weak compositions that result in unsatisfiable guards.

Appendix A

Model listings

Here we present the full versions of some of the models and algorithms referenced throughout the thesis.

```
1  dtmc
2
3  global c0:[0..2]; // requests for slot 1, 0=no requests, 1= 1 request, 2=
       more than one request
4  global c1:[0..2]; // same
5  global c2:[0..2]; // same
6  global c3:[0..2]; // same
7  global c4:[0..2]; // same
8
9  module gateway
10 g: [1..3] init 1;
11 FTR : [0..2] init 0;
12
13 [request_reply] g=1 -> (g' =2); // synchronise with the nodes,relay schedule
14 [] g=2 & c0 = 2 & c1 = 2 & c2 = 2 & c3 = 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
       c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+5-1,2),0)) & (g'=3); //
       generate schedule
15 [] g=2 & c0 = 2 & c1 = 2 & c2 = 2 & c3 < 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
       c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+4-1,2),0)) & (g'=3); //
       generate schedule
16 [] g=2 & c0 = 2 & c1 = 2 & c2 = 2 & c3 = 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
       c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+4-1,2),0)) & (g'=3); //
       generate schedule
17 [] g=2 & c0 = 2 & c1 = 2 & c2 = 2 & c3 < 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
       c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+3-1,2),0)) & (g'=3); //
       generate schedule
18 [] g=2 & c0 = 2 & c1 = 2 & c2 < 2 & c3 = 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
       c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+4-1,2),0)) & (g'=3); //
       generate schedule
```

```
19 [] g=2 & c0 = 2 & c1 = 2 & c2 < 2 & c3 < 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+3-1,2),0)) & (g'=3); //
   generate schedule
20 [] g=2 & c0 = 2 & c1 = 2 & c2 < 2 & c3 = 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+3-1,2),0)) & (g'=3); //
   generate schedule
21 [] g=2 & c0 = 2 & c1 = 2 & c2 < 2 & c3 < 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+2-1,2),0)) & (g'=3); //
   generate schedule
22 [] g=2 & c0 = 2 & c1 < 2 & c2 = 2 & c3 = 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+4-1,2),0)) & (g'=3); //
   generate schedule
23 [] g=2 & c0 = 2 & c1 < 2 & c2 = 2 & c3 < 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+3-1,2),0)) & (g'=3); //
   generate schedule
24 [] g=2 & c0 = 2 & c1 < 2 & c2 = 2 & c3 = 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+3-1,2),0)) & (g'=3); //
   generate schedule
25 [] g=2 & c0 = 2 & c1 < 2 & c2 = 2 & c3 < 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+2-1,2),0)) & (g'=3); //
   generate schedule
26 [] g=2 & c0 = 2 & c1 < 2 & c2 < 2 & c3 = 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+3-1,2),0)) & (g'=3); //
   generate schedule
27 [] g=2 & c0 = 2 & c1 < 2 & c2 < 2 & c3 < 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+2-1,2),0)) & (g'=3); //
   generate schedule
28 [] g=2 & c0 = 2 & c1 < 2 & c2 < 2 & c3 = 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+2-1,2),0)) & (g'=3); //
   generate schedule
29 [] g=2 & c0 = 2 & c1 < 2 & c2 < 2 & c3 < 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+1-1,2),0)) & (g'=3); //
   generate schedule
30
31 [] g=2 & c0 < 2 & c1 = 2 & c2 = 2 & c3 = 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+4-1,2),0)) & (g'=3); //
   generate schedule
32 [] g=2 & c0 < 2 & c1 = 2 & c2 = 2 & c3 < 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+3-1,2),0)) & (g'=3); //
   generate schedule
33 [] g=2 & c0 < 2 & c1 = 2 & c2 = 2 & c3 = 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+3-1,2),0)) & (g'=3); //
   generate schedule
34 [] g=2 & c0 < 2 & c1 = 2 & c2 = 2 & c3 < 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
   c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+2-1,2),0)) & (g'=3); //
   generate schedule
```

```
35 [] g=2 & c0 < 2 & c1 = 2 & c2 < 2 & c3 = 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+3-1,2),0)) & (g'=3); //
     generate schedule
36 [] g=2 & c0 < 2 & c1 = 2 & c2 < 2 & c3 < 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+2-1,2),0)) & (g'=3); //
     generate schedule
37 [] g=2 & c0 < 2 & c1 = 2 & c2 < 2 & c3 = 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+2-1,2),0)) & (g'=3); //
     generate schedule
38 [] g=2 & c0 < 2 & c1 = 2 & c2 < 2 & c3 < 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+1-1,2),0)) & (g'=3); //
     generate schedule
39 [] g=2 & c0 < 2 & c1 < 2 & c2 = 2 & c3 = 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+3-1,2),0)) & (g'=3); //
     generate schedule
40 [] g=2 & c0 < 2 & c1 < 2 & c2 = 2 & c3 < 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+2-1,2),0)) & (g'=3); //
     generate schedule
41 [] g=2 & c0 < 2 & c1 < 2 & c2 = 2 & c3 = 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+2-1,2),0)) & (g'=3); //
     generate schedule
42 [] g=2 & c0 < 2 & c1 < 2 & c2 = 2 & c3 < 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+1-1,2),0)) & (g'=3); //
     generate schedule
43 [] g=2 & c0 < 2 & c1 < 2 & c2 < 2 & c3 = 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+2-1,2),0)) & (g'=3); //
     generate schedule
44 [] g=2 & c0 < 2 & c1 < 2 & c2 < 2 & c3 < 2 & c4 = 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+1-1,2),0)) & (g'=3); //
     generate schedule
45 [] g=2 & c0 < 2 & c1 < 2 & c2 < 2 & c3 = 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+1-1,2),0)) & (g'=3); //
     generate schedule
46 [] g=2 & c0 < 2 & c1 < 2 & c2 < 2 & c3 < 2 & c4 < 2 -> (c0'=0) & (c1'=0) & (
     c2'=0) & (c3'=0) & (c4'=0) & (FTR'=max(min(FTR+0-1,2),0)) & (g'=3); //
     generate schedule
47 [time] g=3 -> (g'=1); // RRM has been generated and broadcast
48 endmodule
49
50
51 //node 1
52 module node1
53   backoff_counter1:[0..2] init 0;
54   slot1 : [-1..5-1] init -1;
55
56   // local state
```

```
57  s1 : [2..8] init 3; // modify this
58  [request_reply] (s1=3) & (backoff_counter1<=1) -> (s1'=7) & (
     backoff_counter1'=0) ; //synchronise, ready to send
59  [request_reply] (s1=3) & (backoff_counter1>1)  -> (s1'=2) & (
     backoff_counter1'=backoff_counter1-1)  ; //count down
60
61  // progress time while backed off
62  [time] s1 = 2 -> (s1'=3) ; // keep progressing through slots
63
64  // randomly choose slot
65  [] s1 = 4 -> 1/5: (slot1'=0) & (s1'=5) + 1/5: (slot1'=1) & (s1'=5) + 1/5:
     (slot1'=2) & (s1'=5)+
66              1/5: (slot1'=3) & (s1'=5) + 1/5: (slot1'=4) & (s1'=5); //
     randomly choose request slot
67
68  // send request
69  [] s1= 5 & slot1 = 0 -> (c0' = min(c0+1,2)) & (s1' = 6);
70  [] s1= 5 & slot1 = 1 -> (c1' = min(c1+1,2)) & (s1' = 6);
71  [] s1= 5 & slot1 = 2 -> (c2' = min(c2+1,2)) & (s1' = 6);
72  [] s1= 5 & slot1 = 3 -> (c3' = min(c3+1,2)) & (s1' = 6);
73  [] s1= 5 & slot1 = 4 -> (c4' = min(c4+1,2)) & (s1' = 6);
74
75  // all sensors have chosen their request slots
76  [time] s1= 7 -> (s1' = 4);
77
78  [request_reply] s1 = 6 & slot1 = 0 & c0 <= 1 -> (s1'=8); //can now send
     data
79  [request_reply] s1 = 6 & slot1 = 1 & c1 <= 1 -> (s1'=8); //can now send
     data
80  [request_reply] s1 = 6 & slot1 = 2 & c2 <= 1 -> (s1'=8); //can now send
     data
81  [request_reply] s1 = 6 & slot1 = 3 & c3 <= 1 -> (s1'=8); //can now send
     data
82  [request_reply] s1 = 6 & slot1 = 4 & c4 <= 1 -> (s1'=8); //can now send
     data
83
84  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 = 2 & c2 = 2 & c3 = 2 &
     c4 = 2 -> (backoff_counter1'=max(min(FTR + 4,2),0)) & (s1'=2); //need to
      retransmit
85  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 = 2 & c2 = 2 & c3 < 2 &
     c4 = 2 -> (backoff_counter1'=max(min(FTR + 3,2),0)) & (s1'=2); //need to
      retransmit
86  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 = 2 & c2 = 2 & c3 = 2 &
     c4 < 2 -> (backoff_counter1'=max(min(FTR + 3,2),0)) & (s1'=2); //need to
      retransmit
```

```
87  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 = 2 & c2 = 2 & c3 < 2 &
     c4 < 2 -> (backoff_counter1'=max(min(FTR + 2,2),0)) & (s1'=2); //need to
      retransmit
88  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 = 2 & c2 < 2 & c3 = 2 &
     c4 = 2 -> (backoff_counter1'=max(min(FTR + 3,2),0)) & (s1'=2); //need to
      retransmit
89  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 = 2 & c2 < 2 & c3 < 2 &
     c4 = 2 -> (backoff_counter1'=max(min(FTR + 2,2),0)) & (s1'=2); //need to
      retransmit
90  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 = 2 & c2 < 2 & c3 = 2 &
     c4 < 2 -> (backoff_counter1'=max(min(FTR + 2,2),0)) & (s1'=2); //need to
      retransmit
91  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 = 2 & c2 < 2 & c3 < 2 &
     c4 < 2 -> (backoff_counter1'=max(min(FTR + 1,2),0)) & (s1'=2); //need to
      retransmit
92  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 < 2 & c2 = 2 & c3 = 2 &
     c4 = 2 -> (backoff_counter1'=max(min(FTR + 3,2),0)) & (s1'=2); //need to
      retransmit
93  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 < 2 & c2 = 2 & c3 < 2 &
     c4 = 2 -> (backoff_counter1'=max(min(FTR + 2,2),0)) & (s1'=2); //need to
      retransmit
94  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 < 2 & c2 = 2 & c3 = 2 &
     c4 < 2 -> (backoff_counter1'=max(min(FTR + 2,2),0)) & (s1'=2); //need to
      retransmit
95  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 < 2 & c2 = 2 & c3 < 2 &
     c4 < 2 -> (backoff_counter1'=max(min(FTR + 1,2),0)) & (s1'=2); //need to
      retransmit
96  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 < 2 & c2 < 2 & c3 = 2 &
     c4 = 2 -> (backoff_counter1'=max(min(FTR + 2,2),0)) & (s1'=2); //need to
      retransmit
97  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 < 2 & c2 < 2 & c3 < 2 &
     c4 = 2 -> (backoff_counter1'=max(min(FTR + 1,2),0)) & (s1'=2); //need to
      retransmit
98  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 < 2 & c2 < 2 & c3 = 2 &
     c4 < 2 -> (backoff_counter1'=max(min(FTR + 1,2),0)) & (s1'=2); //need to
      retransmit
99  [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & c1 < 2 & c2 < 2 & c3 < 2 &
     c4 < 2 -> (backoff_counter1'=max(min(FTR + 0,2),0)) & (s1'=2); //need to
      retransmit
100
101 [request_reply] s1 = 6 & slot1 = 1 & c1 = 2 & c2 = 2 & c3 = 2 & c4 = 2 ->
     (backoff_counter1'=max(min(FTR + 3,2),0)) & (s1'=2); //need to
     retransmit
102 [request_reply] s1 = 6 & slot1 = 1 & c1 = 2 & c2 = 2 & c3 < 2 & c4 = 2 ->
     (backoff_counter1'=max(min(FTR + 2,2),0)) & (s1'=2); //need to
     retransmit
```

```
103   [request_reply] s1 = 6 & slot1 = 1 & c1 = 2 & c2 = 2 & c3 = 2 & c4 < 2 ->
       (backoff_counter1'=max(min(FTR + 2,2),0)) & (s1'=2); //need to
       retransmit
104   [request_reply] s1 = 6 & slot1 = 1 & c1 = 2 & c2 = 2 & c3 < 2 & c4 < 2 ->
       (backoff_counter1'=max(min(FTR + 1,2),0)) & (s1'=2); //need to
       retransmit
105   [request_reply] s1 = 6 & slot1 = 1 & c1 = 2 & c2 < 2 & c3 = 2 & c4 = 2 ->
       (backoff_counter1'=max(min(FTR + 2,2),0)) & (s1'=2); //need to
       retransmit
106   [request_reply] s1 = 6 & slot1 = 1 & c1 = 2 & c2 < 2 & c3 < 2 & c4 = 2 ->
       (backoff_counter1'=max(min(FTR + 1,2),0)) & (s1'=2); //need to
       retransmit
107   [request_reply] s1 = 6 & slot1 = 1 & c1 = 2 & c2 < 2 & c3 = 2 & c4 < 2 ->
       (backoff_counter1'=max(min(FTR + 1,2),0)) & (s1'=2); //need to
       retransmit
108   [request_reply] s1 = 6 & slot1 = 1 & c1 = 2 & c2 < 2 & c3 < 2 & c4 < 2 ->
       (backoff_counter1'=max(min(FTR + 0,2),0)) & (s1'=2); //need to
       retransmit
109
110   [request_reply] s1 = 6 & slot1 = 2 & c2 = 2 & c3 = 2 & c4 = 2 -> (
       backoff_counter1'=max(min(FTR + 2,2),0)) & (s1'=2); //need to retransmit
111   [request_reply] s1 = 6 & slot1 = 2 & c2 = 2 & c3 < 2 & c4 = 2 -> (
       backoff_counter1'=max(min(FTR + 1,2),0)) & (s1'=2); //need to retransmit
112   [request_reply] s1 = 6 & slot1 = 2 & c2 = 2 & c3 = 2 & c4 < 2 -> (
       backoff_counter1'=max(min(FTR + 1,2),0)) & (s1'=2); //need to retransmit
113   [request_reply] s1 = 6 & slot1 = 2 & c2 = 2 & c3 < 2 & c4 < 2 -> (
       backoff_counter1'=max(min(FTR + 0,2),0)) & (s1'=2); //need to retransmit
114
115   [request_reply] s1 = 6 & slot1 = 3 & c3 = 2 & c4=2 -> (backoff_counter1'=
       max(min(FTR+1,2),0)) & (s1'=2); //need to retransmit
116   [request_reply] s1 = 6 & slot1 = 3 & c3 = 2 & c4<2 -> (backoff_counter1'=
       max(min(FTR  ,2),0)) & (s1'=2); //need to retransmit
117   [request_reply] s1 = 6 & slot1 = 4 & c4 = 2 -> (backoff_counter1'=max(min(
       FTR,2),0)) & (s1'=2); //need to retransmit
118
119   [] s1=8 -> (s1'=3);
120 endmodule
121
122 module node2=node1[s1=s2,
123                    backoff_counter1=backoff_counter2,
124                    slot1=slot2]
125 endmodule
```

Listing A.1: Ctrl-MAC model for 5 request slots and 2 sensor devices that has been preprocessed, so that GRIP is able to accept it as input.

```
1 // ctrl_MAC protocol
```

```
dtmc

//add constants
const int TICK = 1; //scaled - 0.05 sec
const int REQUESTSLOT_TIME = 2*TICK; // scaled - 0.1 sec
const int DATASLOT_TIME = 3*TICK; // scaled - 0.15 sec

const int DATA_CHANNELS = 3;
const int SLOTS_PER_CHAN = 3;
const int REQUEST_SLOTS = 5;
const int DATA_SLOT = DATA_CHANNELS*SLOTS_PER_CHAN;
const int TIME_MAX = REQUESTSLOT_TIME*(REQUEST_SLOTS + 1); // length of a
    whole period
const int MAX_FTR = 20; // just a guess at the moment
const int MAX_BACKOFF = 25; // also a guess

// need one of these per request slot
// represent the physical state of the slots
global c0:[0..2]; // requests for slot 1, 0=no requests, 1= 1 request, 2=
    many requests
global c1:[0..2]; // same
global c2:[0..2]; // same
global c3:[0..2]; // same
global c4:[0..2]; // same

module gateway

active1: [0..1] init 1; //if 1 then ready to send request_reply message
active2: [0..1] init 1; //if 1 then not yet decremented FTR
x : [0..TIME_MAX]; //clock for gateway
//local state
g: [1..3] init 1;
//1 at start of request-reply slot
//2 receiving data and request messages
//3  generate schedule
C00 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C10 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C20 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C30 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C40 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
FTR : [0..MAX_FTR] init 0;

[request_reply] g=1 & x = 0 & active1=1 & active2=1-> (active1'=0); //
    synchronise with the nodes, relay schedule
[] g=1 & x=0 & active1=0 & active2=1 -> (FTR'=newFTR) & (active2'=0); //
    decrement FTR if appropriate
```

```
[time] g=1 & active1=0 & active2=0 & x<REQUESTSLOT_TIME-1-> (x'=x+1); //
   increment time through RR slot
[time] g=1 & x=REQUESTSLOT_TIME-1-> (x'=x+1) & (g'=2); //increment time out
   of RR slot
[time] g=2 & (x<TIME_MAX-1) -> (x'=x+1); // move through remaining slots,
   receive data
[time] g=2 & (x=TIME_MAX-1) -> (g'=3); // approach next RR slot
[] g=3 & x>0 -> (x'=0) & (FTR'= min(FTR + new_contentions,MAX_FTR));
[] g=3 & x=0 -> (C00'=c0) & (C10'=c1) & (C20'=c2) & (C30'=c3) & (C40'=c4) &
          (active1'=1) & (active2'=1) & (c0'=0) & (c1'=0) & (c2'=0) & (c3'
   =0) & (c4'=0) & (g'=1); //generate schedule
endmodule


//node 1
module node1
  // clock for station 1
  x1 : [0..TIME_MAX];
  backoff_counter1:[0..MAX_BACKOFF] init 0;
  slot1 : [-1..REQUEST_SLOTS-1] init -1;
  // local state
  s1 : [2..12] init 3; // modify this
  // 1 idle
  // STAGE 2
  // 2 message to send
  // 3 Read feedback_1
  // 4 choose random minislot (MS)
  // 5 start send in MS
  // 6 MS requested
  // 7 collision queue
  // 8 start new MS request
  // STAGE 3
  // 9 Read feedback_2
  // 10 Change frequency
  // 11 Data queue
  // 12 send data
  // 13 complete data transmission

  synchronised1 : bool init false;
  send_data1 : bool init false;

  [request_reply] s1=3 & backoff_counter1 =0 -> (x1' =0) & (s1' =4) & (
   synchronised1' = true); //synchronise, ready to send
  [request_reply] (s1=3) & (backoff_counter1 > 0) -> (s1' =2)  & (
   backoff_counter1'=backoff_counter1 - 1) & (x1'=0); //count down
  // progress time while backed off
```

```
84  [time] s1 = 2 & x1 < TIME_MAX-1 -> (x1' = x1+1); // keep progressing
     through slots
85  [time] s1 = 2 & x1 = TIME_MAX-1 -> (x1' = x1+1) & (s1' = 3); // keep
     progressing through slots
86  // randomly choose slot
87  [] s1 = 4 -> 1/5: (slot1'=0) & (s1'=5)+ 1/5: (slot1'=1) & (s1'=5) + 1/5: (
     slot1'=2) & (s1'=5)+
88              1/5: (slot1'=3) & (s1'=5)+ 1/5: (slot1'=4)& (s1'=5); //
     randomly choose request slot
89  // if time - send request
90  [] s1= 5 & slot1 = 0 & x1 = REQUESTSLOT_TIME ->(c0' = min(c0+1,2)) & (s1'
     =6);
91  [] s1= 5 & slot1 = 1 & x1 = 2*REQUESTSLOT_TIME ->(c1' = min(c1+1,2)) & (s1
     ' = 6);
92  [] s1= 5 & slot1 = 2 & x1 = 3*REQUESTSLOT_TIME -> (c2' = min(c2+1,2))& (s1
     ' = 6);
93  [] s1= 5 & slot1 = 3 & x1 = 4*REQUESTSLOT_TIME -> (c3' = min(c3+1,2))& (s1
     ' = 6);
94  [] s1= 5 & slot1 = 4 & x1 = 5*REQUESTSLOT_TIME -> (c4' = min(c4+1,2))& (s1
     ' = 6);
95  // if not time to send message, move on
96  [time] s1 = 5 & slot1 >= 0 & x1 < slot1*REQUESTSLOT_TIME +
     REQUESTSLOT_TIME -> (s1'=5) & (x1'=x1+1);
97  // message is sent, keep progressing through the slots
98  [time] s1 = 6 & x1 < TIME_MAX -> (x1' = x1+1); // keep progressing through
      slots
99
100 [request_reply] s1 = 6 & x1 = TIME_MAX & slot1 = 0 & C00 <= 1 -> (x1'=0) &
      (s1'=12) & (slot1'=-1); //can now send data
101 [request_reply] s1 = 6 & x1 = TIME_MAX & slot1 = 1 & C10 <= 1 -> (x1'=0) &
      (s1'=12) & (slot1'=-1); //can now send data
102 [request_reply] s1 = 6 & x1 = TIME_MAX & slot1 = 2 & C20 <= 1 -> (x1'=0) &
      (s1'=12) & (slot1'=-1); //can now send data
103 [request_reply] s1 = 6 & x1 = TIME_MAX & slot1 = 3 & C30 <= 1 -> (x1'=0) &
      (s1'=12) & (slot1'=-1); //can now send data
104 [request_reply] s1 = 6 & x1 = TIME_MAX & slot1 = 4 & C40 <= 1 -> (x1'=0) &
      (s1'=12) & (slot1'=-1); //can now send data
105
106 [request_reply] s1 = 6 & x1 = TIME_MAX & slot1 = 0 & C00 = 2 -> (x1'=0) &
     (backoff_counter1'=backoff0) & (s1'=2) & (slot1'=-1); //need to
     retransmit
107 [request_reply] s1 = 6 & x1 = TIME_MAX & slot1 = 1 & C10 = 2 -> (x1'=0) &
     (backoff_counter1'=backoff1) & (s1'=2) & (slot1'=-1); //need to
     retransmit
108 [request_reply] s1 = 6 & x1 = TIME_MAX & slot1 = 2 & C20 = 2 -> (x1'=0) &
     (backoff_counter1'=backoff2) & (s1'=2) & (slot1'=-1); //need to
```

```
    retransmit
109   [request_reply] s1 = 6 & x1 = TIME_MAX & slot1 = 3 & C30 = 2 -> (x1'=0) &
    (backoff_counter1'=backoff3) & (s1'=2) & (slot1'=-1); //need to
    retransmit
110   [request_reply] s1 = 6 & x1 = TIME_MAX & slot1 = 4 & C40 = 2 -> (x1'=0) &
    (backoff_counter1'=backoff4) & (s1'=2) & (slot1'=-1); //need to
    retransmit
111
112   [request_reply] s1 = 7 -> (s1'=7); // collision // This state is never
    reached; Delete?
113
114   [end] s1=12 -> (send_data1' = true); // ready to send data, for now just
    end here
115   // progress time at the end
116   [time] s1 = 12 & x1 < TIME_MAX -> (x1' = x1+1); // keep progressing
    through time slots
117   [request_reply] s1 = 12 & x1 = TIME_MAX -> (x1'=0);  // This sensor has
    successfully sent its message; for now just listen to future RRs and do
    nothing
118 endmodule
119
120 module node2=node1[x1=x2,
121                    s1=s2,
122                    backoff_counter1=backoff_counter2,
123                    slot1=slot2,
124                    synchronised1=synchronised2,
125                    send_data1=send_data2]
126 endmodule
127
128 module node3=node1[x1=x3,
129                    s1=s3,
130                    backoff_counter1=backoff_counter3,
131                    slot1=slot3,
132                    synchronised1=synchronised3,
133                    send_data1=send_data3]
134 endmodule
135
136 module node4=node1[x1=x4,
137                    s1=s4,
138                    backoff_counter1=backoff_counter4,
139                    slot1=slot4,
140                    synchronised1=synchronised4,
141                    send_data1=send_data4]
142 endmodule
143
144 module node5=node1[x1=x5,
```

```
                    s1=s5,
                    backoff_counter1=backoff_counter5,
                    slot1=slot5,
                    synchronised1=synchronised5,
                    send_data1=send_data5]
endmodule

module node6=node1[x1=x6,
                    s1=s6,
                    backoff_counter1=backoff_counter6,
                    slot1=slot6,
                    synchronised1=synchronised6,
                    send_data1=send_data6]
endmodule

module node7=node1[x1=x7,
                    s1=s7,
                    backoff_counter1=backoff_counter7,
                    slot1=slot7,
                    synchronised1=synchronised7,
                    send_data1=send_data7]
endmodule

module node8=node1[x1=x8,
                    s1=s8,
                    backoff_counter1=backoff_counter8,
                    slot1=slot8,
                    synchronised1=synchronised8,
                    send_data1=send_data8]
endmodule


formula new_contentions = ((C00=2?1:0) + (C10=2?1:0) + (C20=2?1:0) + (C30
    =2?1:0) + (C40=2?1:0));

//formulas assume p starts from 0

//one per request slot
formula position0 =  0;   // position of slot 0 among contention slots in
    this round
formula position1 =  (C00=2?1:0);   // position of slot 1 among contention
    slots in this round
formula position2 =  (C00=2?1:0) + (C10=2?1:0); // position of slot 2 among
    contention slots in this round
formula position3 =  (C00=2?1:0) + (C10=2?1:0) + (C20=2?1:0); // position of
     slot 3 among contention slots in this round
```

```
formula position4 =  (C00=2?1:0) + (C10=2?1:0) + (C20=2?1:0) + (C30=2?1:0);
   // position of slot 4 among contention slots in this round

formula newFTR = (FTR>0?FTR-1:FTR);


//one per request slot);
formula backoff0 = min(FTR + new_contentions-position0,MAX_BACKOFF);
formula backoff1 = min(FTR + new_contentions-position1,MAX_BACKOFF);
formula backoff2 = min(FTR + new_contentions-position2,MAX_BACKOFF);
formula backoff3 = min(FTR + new_contentions-position3,MAX_BACKOFF);
formula backoff4 = min(FTR + new_contentions-position4,MAX_BACKOFF);

// labels
label "firstSynch1" = synchronised1=true;
label "sent1" = send_data1=true;
```

Listing A.2: PRISM code for model 1

```
 // ctrl_MAC protocol
dtmc

//add constants
const int REQUEST_SLOTS = 5;
const int MAX_FTR = 2; // just a guess at the moment
const int MAX_BACKOFF = 2; // also a guess

// need one of these per request slot
// represent the physical state of the slots
global c0:[0..2]; // requests for slot 1, 0=no requests, 1= 1 request, 2=
   more than one request
global c1:[0..2]; // same
global c2:[0..2]; // same
global c3:[0..2]; // same
global c4:[0..2]; // same

module gateway
//1 at start of request-reply slot
//2 receiving data and request messages
//3  generate schedule
g: [1..4] init 1;

C00 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C10 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C20 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C30 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
C40 : [0..2]; // 0 - no request, 1 - no contention, 2 - contention
```

```
28 FTR : [0..MAX_FTR] init 0;
29
30 [request_reply] g=1 -> (g' =2); // synchronise with the nodes,relay schedule
31 [] g=2 -> (g'=3);
32 [time] g=3 -> (g'=2); // sensors have chosen request slots, can generate RRM
33 [] g=4 -> (C00'=c0) & (C10'=c1) & (C20'=c2) & (C30'=c3) & (C40'=c4) &
34           (c0'=0) & (c1'=0) & (c2'=0) & (c3'=0) & (c4'=0) &    // generate
35           (FTR'=min(newFTR+new_contentions,MAX_FTR)) & (g'=1); // schedule
36 endmodule
37
38
39 //node 1
40 module node1
41   backoff_counter1:[0..MAX_BACKOFF] init 0;
42   slot1 : [-1..REQUEST_SLOTS-1] init -1;
43
44   // local state
45   s1 : [2..12] init 3; // modify this
46
47   [request_reply] (s1=3) & (backoff_counter1=0) -> (s1'=4) ; //synchronise,
      ready to send
48   [request_reply] (s1=3) & (backoff_counter1>0) -> (s1'=2) & (
      backoff_counter1'=backoff_counter1-1) ; //count down
49
50   // progress time while backed off
51   [time] s1 = 2 -> (s1' = 3) ; // keep progressing through slots
52
53   // randomly choose slot
54   [] s1 = 4 -> 1/5: (slot1' = 0) & (s1'=5)+ 1/5: (slot1' = 1) & (s1'=5) +
      1/5: (slot1' = 2) & (s1'=5)+
55               1/5: (slot1' = 3) & (s1'=5)+ 1/5: (slot1'=4)& (s1'=5); //
      randomly choose request slot
56
57   // send request
58   [] s1= 5 & slot1 = 0 -> (c0' = min(c0+1,2)) & (s1' = 7);
59   [] s1= 5 & slot1 = 1 -> (c1' = min(c1+1,2)) & (s1' = 7);
60   [] s1= 5 & slot1 = 2 -> (c2' = min(c2+1,2)) & (s1' = 7);
61   [] s1= 5 & slot1 = 3 -> (c3' = min(c3+1,2)) & (s1' = 7);
62   [] s1= 5 & slot1 = 4 -> (c4' = min(c4+1,2)) & (s1' = 7);
63
64   // all sensors have chosen their request slots
65   [time] s1= 7 -> (s1' = 6);
66
67   [request_reply] s1 = 6 & slot1 = 0 & C00 <= 1 -> (s1'=12) & (slot1'=-1);
      //can now send data
68   [request_reply] s1 = 6 & slot1 = 1 & C10 <= 1 -> (s1'=12) & (slot1'=-1);
```

```
    //can now send data
69  [request_reply] s1 = 6 & slot1 = 2 & C20 <= 1 -> (s1'=12) & (slot1'=-1);
    //can now send data
70  [request_reply] s1 = 6 & slot1 = 3 & C30 <= 1 -> (s1'=12) & (slot1'=-1);
    //can now send data
71  [request_reply] s1 = 6 & slot1 = 4 & C40 <= 1 -> (s1'=12) & (slot1'=-1);
    //can now send data
72
73  [request_reply] s1 = 6 & slot1 = 0 & C00 = 2 & backoff0>=0 -> (
    backoff_counter1'=backoff0) & (s1'=((backoff0=0)?4:2)) & (slot1'=-1); //
    need to retransmit
74  [request_reply] s1 = 6 & slot1 = 1 & C10 = 2 & backoff1>=0 -> (
    backoff_counter1'=backoff1) & (s1'=((backoff1=0)?4:2)) & (slot1'=-1); //
    need to retransmit
75  [request_reply] s1 = 6 & slot1 = 2 & C20 = 2 & backoff2>=0 -> (
    backoff_counter1'=backoff2) & (s1'=((backoff2=0)?4:2)) & (slot1'=-1); //
    need to retransmit
76  [request_reply] s1 = 6 & slot1 = 3 & C30 = 2 & backoff3>=0 -> (
    backoff_counter1'=backoff3) & (s1'=((backoff3=0)?4:2)) & (slot1'=-1); //
    need to retransmit
77  [request_reply] s1 = 6 & slot1 = 4 & C40 = 2 & backoff4>=0 -> (
    backoff_counter1'=backoff4) & (s1'=((backoff4=0)?4:2)) & (slot1'=-1); //
    need to retransmit
78
79  [time] s1=12 -> (s1'=12);
80  [request_reply] s1=12 -> (s1'=12);
81 endmodule
82
83 module node2=node1[s1=s2,
84                    backoff_counter1=backoff_counter2,
85                    slot1=slot2]
86 endmodule
87
88 module node3=node1[s1=s3,
89                    backoff_counter1=backoff_counter3,
90                    slot1=slot3]
91 endmodule
92
93 module node4=node1[s1=s4,
94                   backoff_counter1=backoff_counter4,
95                   slot1=slot4]
96 endmodule
97
98 module node5=node1[s1=s5,
99                   backoff_counter1=backoff_counter5,
100                  slot1=slot5]
```

```
endmodule

module node6=node1[s1=s6,
                   backoff_counter1=backoff_counter6,
                   slot1=slot6]
endmodule

module node7=node1[s1=s7,
                   backoff_counter1=backoff_counter7,
                   slot1=slot7]
endmodule

// Use this only for the new FTR calculation in the generation of a schedule
formula new_contentions = ((c0=2?1:0) + (c1=2?1:0) + (c2=2?1:0) + (c3=2?1:0)
    + (c4=2?1:0));

formula position0 =  0;   // position of slot 0 among contention slots in
   this round
formula position1 =  (C00=2?1:0);   // position of slot 1 among contention
   slots in this round
formula position2 =  (C00=2?1:0) + (C10=2?1:0); // position of slot 2 among
   contention slots in this round
formula position3 =  (C00=2?1:0) + (C10=2?1:0) + (C20=2?1:0); // position of
    slot 3 among contention slots in this round
formula position4 =  (C00=2?1:0) + (C10=2?1:0) + (C20=2?1:0) + (C30=2?1:0);
   // position of slot 4 among contention slots in this round

formula newFTR = (FTR>0?FTR-1:FTR);

formula backoff0 = min(FTR - (new_contentions-position0),MAX_BACKOFF);
formula backoff1 = min(FTR - (new_contentions-position1),MAX_BACKOFF);
formula backoff2 = min(FTR - (new_contentions-position2),MAX_BACKOFF);
formula backoff3 = min(FTR - (new_contentions-position3),MAX_BACKOFF);
formula backoff4 = min(FTR - (new_contentions-position4),MAX_BACKOFF);
```

Listing A.3: PRISM code for model 2

```
 // ctrl_MAC protocol
dtmc

//add constants
const int REQUEST_SLOTS = 5;
const int MAX_FTR = 2; // just a guess at the moment
const int MAX_BACKOFF = 2; // also a guess

// need one of these per request slot
// represent the physical state of the slots
```

```
11 global c0:[0..2]; // requests for slot 1, 0=no requests, 1= 1 request, 2=
       more than one request
12 global c1:[0..2]; // same
13 global c2:[0..2]; // same
14 global c3:[0..2]; // same
15 global c4:[0..2]; // same
16
17 module gateway
18 //1 at start of request-reply slot
19 //2 generate schedule
20 //3 receiving data and request messages
21 g: [1..3] init 1;
22
23 FTR : [0..MAX_FTR] init 0;
24
25 [request_reply] g=1 -> (g' =2); // synchronise with the nodes,relay schedule
26 [] g=2 -> (c0'=0) & (c1'=0) & (c2'=0) & (c3'=0) & (c4'=0) &    // generate
27           (FTR'=newFTR) & (g'=3);                              // schedule
28 [time] g=3 -> (g'=1); // RRM has been generated and broadcast
29 endmodule
30
31
32 //node 1
33 module node1
34   backoff_counter1:[0..MAX_BACKOFF] init 0;
35   slot1 : [-1..REQUEST_SLOTS-1] init -1;
36
37   // local state
38   s1 : [2..8] init 3; // modify this
39
40   [request_reply] (s1=3) & (backoff_counter1<=1) -> (s1'=7) & (
       backoff_counter1'=0) ; //synchronise, ready to send
41   [request_reply] (s1=3) & (backoff_counter1>1)  -> (s1'=2) & (
       backoff_counter1'=backoff_counter1-1)  ; //count down
42
43   // progress time while backed off
44   [time] s1 = 2 -> (s1'=3) ; // keep progressing through slots
45
46   // randomly choose slot
47   [] s1 = 4 -> 1/5: (slot1'=0) & (s1'=5) + 1/5: (slot1'=1) & (s1'=5) + 1/5:
       (slot1'=2) & (s1'=5)+
48               1/5: (slot1'=3) & (s1'=5) + 1/5: (slot1'=4) & (s1'=5); //
       randomly choose request slot
49
50   // send request
51   [] s1= 5 & slot1 = 0 -> (c0' = min(c0+1,2)) & (s1' = 6);
```

```
52   [] s1= 5 & slot1 = 1 -> (c1' = min(c1+1,2)) & (s1' = 6);
53   [] s1= 5 & slot1 = 2 -> (c2' = min(c2+1,2)) & (s1' = 6);
54   [] s1= 5 & slot1 = 3 -> (c3' = min(c3+1,2)) & (s1' = 6);
55   [] s1= 5 & slot1 = 4 -> (c4' = min(c4+1,2)) & (s1' = 6);
56
57   // all sensors have chosen their request slots
58   [time] s1= 7 -> (s1' = 4);
59
60   [request_reply] s1 = 6 & slot1 = 0 & c0 <= 1 -> (s1'=8) & (slot1'=-1); //
      can now send data
61   [request_reply] s1 = 6 & slot1 = 1 & c1 <= 1 -> (s1'=8) & (slot1'=-1); //
      can now send data
62   [request_reply] s1 = 6 & slot1 = 2 & c2 <= 1 -> (s1'=8) & (slot1'=-1); //
      can now send data
63   [request_reply] s1 = 6 & slot1 = 3 & c3 <= 1 -> (s1'=8) & (slot1'=-1); //
      can now send data
64   [request_reply] s1 = 6 & slot1 = 4 & c4 <= 1 -> (s1'=8) & (slot1'=-1); //
      can now send data
65
66   [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & backoff0>=0 -> (
      backoff_counter1'=backoff0) & (s1'=((backoff0=0)?7:2)) & (slot1'=-1); //
      need to retransmit
67   [request_reply] s1 = 6 & slot1 = 1 & c1 = 2 & backoff1>=0 -> (
      backoff_counter1'=backoff1) & (s1'=((backoff1=0)?7:2)) & (slot1'=-1); //
      need to retransmit
68   [request_reply] s1 = 6 & slot1 = 2 & c2 = 2 & backoff2>=0 -> (
      backoff_counter1'=backoff2) & (s1'=((backoff2=0)?7:2)) & (slot1'=-1); //
      need to retransmit
69   [request_reply] s1 = 6 & slot1 = 3 & c3 = 2 & backoff3>=0 -> (
      backoff_counter1'=backoff3) & (s1'=((backoff3=0)?7:2)) & (slot1'=-1); //
      need to retransmit
70   [request_reply] s1 = 6 & slot1 = 4 & c4 = 2 & backoff4>=0 -> (
      backoff_counter1'=backoff4) & (s1'=((backoff4=0)?7:2)) & (slot1'=-1); //
      need to retransmit
71
72   [time] s1=8 -> (s1'=8);
73   [request_reply] s1=8 -> (s1'=8);
74 endmodule
75
76 module node2=node1[s1=s2,
77                    backoff_counter1=backoff_counter2,
78                    slot1=slot2]
79 endmodule
80
81 module node3=node1[s1=s3,
82                    backoff_counter1=backoff_counter3,
```

```
                    slot1=slot3]
endmodule

module node4=node1[s1=s4,
                   backoff_counter1=backoff_counter4,
                   slot1=slot4]
endmodule

module node5=node1[s1=s5,
                   backoff_counter1=backoff_counter5,
                   slot1=slot5]
endmodule

module node6=node1[s1=s6,
                   backoff_counter1=backoff_counter6,
                   slot1=slot6]
endmodule

module node7=node1[s1=s7,
                   backoff_counter1=backoff_counter7,
                   slot1=slot7]
endmodule

module node8=node1[s1=s8,
                   backoff_counter1=backoff_counter8,
                   slot1=slot8]
endmodule

module node9=node1[s1=s9,
                   backoff_counter1=backoff_counter9,
                   slot1=slot9]
endmodule

module node10=node1[s1=s10,
                   backoff_counter1=backoff_counter10,
                   slot1=slot10]
endmodule

// Note that these are ALL contentions so (r+1)
formula new_contentions = ((c0=2?1:0) + (c1=2?1:0) + (c2=2?1:0) + (c3=2?1:0)
     + (c4=2?1:0));

formula position0 =  0;   // position of slot 0 among contention slots in
    this round
formula position1 =  (c0=2?1:0);   // position of slot 1 among contention
    slots in this round
```

```
126 formula position2 =  (c0=2?1:0) + (c1=2?1:0); // position of slot 2 among
       contention slots in this round
127 formula position3 =  (c0=2?1:0) + (c1=2?1:0) + (c2=2?1:0); // position of
       slot 3 among contention slots in this round
128 formula position4 =  (c0=2?1:0) + (c1=2?1:0) + (c2=2?1:0) + (c3=2?1:0); //
       position of slot 4 among contention slots in this round
129
130 formula newFTR = max(min(FTR+new_contentions-1,MAX_FTR),0);
131
132 //one per request slot);
133 formula backoff0 = max(min(FTR + new_contentions-1-position0,MAX_BACKOFF),0)
       ;
134 formula backoff1 = max(min(FTR + new_contentions-1-position1,MAX_BACKOFF),0)
       ;
135 formula backoff2 = max(min(FTR + new_contentions-1-position2,MAX_BACKOFF),0)
       ;
136 formula backoff3 = max(min(FTR + new_contentions-1-position3,MAX_BACKOFF),0)
       ;
137 formula backoff4 = max(min(FTR + new_contentions-1-position4,MAX_BACKOFF),0)
       ;
```

Listing A.4: PRISM code for model 3

```
1  // ctrl_MAC protocol
2 dtmc
3
4 //add constants
5 const int TICK = 1; //scaled - 0.05 sec
6 const int REQUESTSLOT_TIME = 2*TICK; // scaled - 0.1 sec
7 const int DATASLOT_TIME = 3*TICK; // scaled - 0.15 sec
8
9 const int DATA_CHANNELS = 3;
10 const int SLOTS_PER_CHAN = 3;
11 const int REQUEST_SLOTS = 5;
12 const int DATA_SLOT = DATA_CHANNELS*SLOTS_PER_CHAN;
13 // number of sensors = 6
14 const int TOTAL_SENSORS = 5;
15 const int POPULATION = TOTAL_SENSORS-1;
16 const int MAX_FTR = 2; // this should be floor(number of sensors/2)
17 const int MAX_BACKOFF = 2; // This value should either be he same as MAX_FTR
       or MAX_FTR+1
18
19 // need one of these per request slot
20 // represent the physical state of the slots
21
22 global c0:[0..2]; // requests for slot 1, number of requests
23 global c1:[0..2]; // same
```

```
24 global c2:[0..2]; // same
25 global c3:[0..2]; // same
26 global c4:[0..2]; // same
27
28 module gateway
29 g: [1..4] init 1;
30 FTR : [0..MAX_FTR] init 0;
31 // population counters
32 idle: [0..POPULATION] init POPULATION;// idle sensors
33 sending: [0..POPULATION] init 0;    // sensors sending requests this round
34 send0: [0..POPULATION] init 0;      // sensors having chosen request slot 0
35 send1: [0..POPULATION] init 0;      // sensors having chosen request slot 1
36 send2: [0..POPULATION] init 0;      // sensors having chosen request slot 2
37 send3: [0..POPULATION] init 0;      // sensors having chosen request slot 3
38 send4: [0..POPULATION] init 0;      // sensors having chosen request slot 4
39 backed_off1: [0..POPULATION] init 0;// sensors backed off for 1 RR cycle
40 backed_off2: [0..POPULATION] init 0;// sensors backed off for 2 RR cycles
41
42 [] g=1 & idle>0 & sending>=0-> (idle'=0) & (sending'=min(sending+idle,
      POPULATION)); // transition to attempt to send
43 // each sending sensor chooses the slot they like
44 [] g=1 & idle=0 & sending>0 ->
45 1/5:(sending'=sending-1)&(send0'=min(send0+1,POPULATION))&(c0'=min(c0+1,2))+
46 1/5:(sending'=sending-1)&(send1'=min(send1+1,POPULATION))&(c1'=min(c1+1,2))+
47 1/5:(sending'=sending-1)&(send2'=min(send2+1,POPULATION))&(c2'=min(c2+1,2))+
48 1/5:(sending'=sending-1)&(send3'=min(send3+1,POPULATION))&(c3'=min(c3+1,2))+
49 1/5:(sending'=sending-1)&(send4'=min(send4+1,POPULATION))&(c4'=min(c4+1,2));
50 // every sensor has chosen a slot
51 [time] g=1 & idle=0 & sending=0 -> (g'=2);
52 // change populations to specific backoff population based on congestion and
       current FTR
53 [] g=2 & c0>1 & send0>0 & (FTR+new_contentions-position0=1)->(send0'=0)&(
      backed_off1'=min(backed_off1+send0,POPULATION));
54 [] g=2 & c0>1 & send0>0 & (FTR+new_contentions-position0=2)->(send0'=0)&(
      backed_off2'=min(backed_off2+send0,POPULATION));
55
56 [] g=2 & c1>1 & send1>0 & (FTR+new_contentions-position1=1)->(send1'=0)&(
      backed_off1'=min(backed_off1+send1,POPULATION));
57 [] g=2 & c1>1 & send1>0 & (FTR+new_contentions-position1=2)->(send1'=0)&(
      backed_off2'=min(backed_off2+send1,POPULATION));
58
59 [] g=2 & c2>1 & send2>0 & (FTR+new_contentions-position2=1)->(send2'=0)&(
      backed_off1'=min(backed_off1+send2,POPULATION));
60 [] g=2 & c2>1 & send2>0 & (FTR+new_contentions-position2=2)->(send2'=0)&(
      backed_off2'=min(backed_off2+send2,POPULATION));
61
```

```
62 [] g=2 & c3>1 & send3>0 & (FTR+new_contentions-position3=1)->(send3'=0)&(
     backed_off1'=min(backed_off1+send3,POPULATION));
63 [] g=2 & c3>1 & send3>0 & (FTR+new_contentions-position3=2)->(send3'=0)&(
     backed_off2'=min(backed_off2+send3,POPULATION));
64
65 [] g=2 & c4>1 & send4>0 & (FTR+new_contentions-position4=1)->(send4'=0)&(
     backed_off1'=min(backed_off1+send4,POPULATION));
66 [] g=2 & c4>1 & send4>0 & (FTR+new_contentions-position4=2)->(send4'=0)&(
     backed_off2'=min(backed_off2+send4,POPULATION));
67 // change population to idle if transmission was successful
68 [] g=2 & c0=1 -> (idle'=min(idle + send0,POPULATION)) & (send0'=0);
69 [] g=2 & c1=1 -> (idle'=min(idle + send1,POPULATION)) & (send1'=0);
70 [] g=2 & c2=1 -> (idle'=min(idle + send2,POPULATION)) & (send2'=0);
71 [] g=2 & c3=1 -> (idle'=min(idle + send3,POPULATION)) & (send3'=0);
72 [] g=2 & c4=1 -> (idle'=min(idle + send4,POPULATION)) & (send4'=0);
73 // all requests have been sorted; make RR
74 [request_reply] g=2 & send0+send1+send2+send3+send4=0 -> (g'=3)
75   & (sending'=min(sending+backed_off1,POPULATION))
76   & (backed_off1'=backed_off2) & (backed_off2'=0);
77 // reset request slot counters
78 [] g=3 -> (c0'=0) & (c1'=0) & (c2'=0) & (c3'=0) & (c4'=0) &    // generate
79           (FTR'=newFTR) & (g'=4);                              // schedule
80 [rr_end] g=4 -> (g'=1);
81   endmodule
82
83 //node 1
84 module node1
85   backoff_counter1:[0..MAX_BACKOFF] init 0;
86   slot1 : [-1..REQUEST_SLOTS-1] init -1;
87
88   // local state
89   s1 : [1..8] init 2; // modify this
90
91   [request_reply] (s1=3) & (backoff_counter1<=1) -> (s1'=7) & (
      backoff_counter1'=0) ; //synchronise, ready to send
92   [request_reply] (s1=3) & (backoff_counter1>1)  -> (s1'=2) & (
      backoff_counter1'=backoff_counter1-1)  ; //count down
93
94   // progress time while backed off
95   [rr_end] s1 = 2 -> (s1'=2) ; // keep progressing through slots
96   [time] s1 = 2 -> (s1'=3) ; // keep progressing through slots
97
98   // randomly choose slot
99   [] s1 = 4 -> 1/5: (slot1'=0) & (s1'=5) + 1/5: (slot1'=1) & (s1'=5) + 1/5:
      (slot1'=2) & (s1'=5)+
100              1/5: (slot1'=3) & (s1'=5) + 1/5: (slot1'=4) & (s1'=5); //
```

```
    randomly choose request slot
101
102   // send request
103   [] s1=5 & slot1=0 -> (c0' = min(c0+1,2)) & (s1' = 1);
104   [] s1=5 & slot1=1 -> (c1' = min(c1+1,2)) & (s1' = 1);
105   [] s1=5 & slot1=2 -> (c2' = min(c2+1,2)) & (s1' = 1);
106   [] s1=5 & slot1=3 -> (c3' = min(c3+1,2)) & (s1' = 1);
107   [] s1=5 & slot1=4 -> (c4' = min(c4+1,2)) & (s1' = 1);
108
109   // all sensors have chosen their request slots
110   [rr_end] s1=7 -> (s1'=4);
111   [time] s1=1 -> (s1'=6);
112
113   [request_reply] s1 = 6 & slot1 = 0 & c0 <= 1 -> (s1'=8) & (slot1'=-1); //
       can now send data
114   [request_reply] s1 = 6 & slot1 = 1 & c1 <= 1 -> (s1'=8) & (slot1'=-1); //
       can now send data
115   [request_reply] s1 = 6 & slot1 = 2 & c2 <= 1 -> (s1'=8) & (slot1'=-1); //
       can now send data
116   [request_reply] s1 = 6 & slot1 = 3 & c3 <= 1 -> (s1'=8) & (slot1'=-1); //
       can now send data
117   [request_reply] s1 = 6 & slot1 = 4 & c4 <= 1 -> (s1'=8) & (slot1'=-1); //
       can now send data
118
119   [request_reply] s1 = 6 & slot1 = 0 & c0 = 2 & backoff0>=0 -> (
       backoff_counter1'=backoff0) & (s1'=((backoff0=0)?7:2)) & (slot1'=-1); //
       need to retransmit
120   [request_reply] s1 = 6 & slot1 = 1 & c1 = 2 & backoff1>=0 -> (
       backoff_counter1'=backoff1) & (s1'=((backoff1=0)?7:2)) & (slot1'=-1); //
       need to retransmit
121   [request_reply] s1 = 6 & slot1 = 2 & c2 = 2 & backoff2>=0 -> (
       backoff_counter1'=backoff2) & (s1'=((backoff2=0)?7:2)) & (slot1'=-1); //
       need to retransmit
122   [request_reply] s1 = 6 & slot1 = 3 & c3 = 2 & backoff3>=0 -> (
       backoff_counter1'=backoff3) & (s1'=((backoff3=0)?7:2)) & (slot1'=-1); //
       need to retransmit
123   [request_reply] s1 = 6 & slot1 = 4 & c4 = 2 & backoff4>=0 -> (
       backoff_counter1'=backoff4) & (s1'=((backoff4=0)?7:2)) & (slot1'=-1); //
       need to retransmit
124
125   [request_reply] s1=8 -> (s1'=8);
126   [rr_end] s1=8 -> (s1'=8);
127   [time] s1=8 -> (s1'=8);
128 endmodule
129
130 formula new_contentions = ((c0=2?1:0) + (c1=2?1:0) + (c2=2?1:0) + (c3=2?1:0)
```

```
    + (c4=2?1:0));
131
132 //one per request slot
133 formula position0 =  0;   // position of slot 0 among contention slots in
    this round
134 formula position1 =  (c0=2?1:0);   // position of slot 1 among contention
    slots in this round
135 formula position2 =  (c0=2?1:0) + (c1=2?1:0); // position of slot 2 among
    contention slots in this round
136 formula position3 =  (c0=2?1:0) + (c1=2?1:0) + (c2=2?1:0); // position of
    slot 3 among contention slots in this round
137 formula position4 =  (c0=2?1:0) + (c1=2?1:0) + (c2=2?1:0) + (c3=2?1:0); //
    position of slot 4 among contention slots in this round
138
139 formula newFTR = (FTR>0?FTR-1:FTR);
140
141
142 //one per request slot);
143 formula backoff0 = min(FTR + new_contentions-position0,MAX_BACKOFF);
144 formula backoff1 = min(FTR + new_contentions-position1,MAX_BACKOFF);
145 formula backoff2 = min(FTR + new_contentions-position2,MAX_BACKOFF);
146 formula backoff3 = min(FTR + new_contentions-position3,MAX_BACKOFF);
147 formula backoff4 = min(FTR + new_contentions-position4,MAX_BACKOFF);
```

Listing A.5: PRISM code for model 4

Appendix B

List of Terms

Acronyms

BDD Binary Decision Diagram. 11, 15, 17

CDMA Code Division Multiple Access. 23

CPS Cyber-Physical System. 2, 3, 12, 24, 53, 117

CTMC Continious Time Markov Chain. 81

DTMC Discrete Time Markov Chain. 28, 32, 33, 35, 39, 53, 81, 117

FDMA Frequency Division Multiple Access. 23

FTR Failed Transmission Requests. viii, 49, 50, 54, 57, 58, 61, 63, 64, 67, 68, 74, 75, 112

LPWA Low-Power Wide Area. 4, 5, 24, 46, 47

MDP Markov Decision Process. 17, 33, 39–42, 81, 117

RRC Request-Reply Cycle. v, 47, 50–52, 54–61, 65, 69, 70, 73–75, 77–79, 112, 119

RRM Request-Reply Message. 47–52, 54, 56–58, 61, 63, 68–70, 73, 74, 76, 78, 111, 118

S4 Science of Sensor Systems Software. 4, 46

TDMA Time Division Multiple Access. 23, 24, 52

WA-CPS Wide Area Cyber-Physical System. 3, 4, 24, 46, 47

WSN Wireless sensor network. vii, 20–24, 26, 27, 117

Bibliography

[1] Muhammad Umar Aftab et al. “A Review Study of Wireless Sensor Networks and Its Security”. In: *Communications and Network* 7 (2015), pp. 172–179.

[2] Sheldon B. Akers. “Binary Decision Diagrams”. In: *IEEE Transactions on Computers* C-27 (1978), pp. 509–516.

[3] Kemal Akkaya and Mohamed Younis. “A survey on routing protocols for wireless sensor networks”. In: *Ad hoc networks* 3.3 (2005), pp. 325–349.

[4] I.F. Akyildiz et al. “A survey on sensor networks”. In: *IEEE Communications Magazine* 40.8 (2002), pp. 102–114. DOI: 10.1109/MCOM.2002.1024422.

[5] R. Alur, R. Brayton, et al. “Partial-Order Reduction in Symbolic State Space Exploration”. In: vol. 1254. Apr. 2006, pp. 340–351. ISBN: 978-3-540-63166-8. DOI: 10.1007/3-540-63166-6_34.

[6] Rajeev Alur, Costas A. Courcoubetis, and Thomas A. Henzinger. “Computing Accumulated Delays in Real-time Systems”. In: *Formal Methods in System Design* 11 (1993), pp. 137–155.

[7] Rajeev Alur and Thomas A. Henzinger. “Reactive Modules”. In: *Formal Methods in System Design* 15 (1996), pp. 7–48.

[8] Dana Angluin et al. “Computation in Networks of Passively Mobile Finite-State Sensors”. In: *Distributed Computing*. ACM Press, 2004, pp. 290–299.

[9] Blair Archibald, Géza Kulcsár, and Michele Sevegnani. “A Tale of Two Graph Models: A Case Study in Wireless Sensor Networks”. In: *Form. Asp. Comput.* 33.6 (Dec. 2021), pp. 1249–1277. ISSN: 0934-5043. DOI: 10.1007/s00165-021-00558-z. URL: https://doi.org/10.1007/s00165-021-00558-z.

[10] J. Aspnes and M. Herlihy. “Fast Randomized Consensus Using Shared Memory”. In: *Journal of Algorithms* 15.1 (1990), pp. 441–460.

[11] Mauricio Ayala-Rincón and César A. Muñoz. “Interactive Theorem Proving: 8th International Conference, ITP 2017, Brasília, Brazil, September 26–29, 2017, Proceedings”. In: *Interactive Theorem Proving* (2017).

[12] Tomáš Babiak et al. “The Hanoi Omega-Automata Format”. In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 479–486. ISBN: 978-3-319-21690-4.

[13] Hamid Bagheri et al. “Detection of Design Flaws in the Android Permission Protocol Through Bounded Verification”. In: (Jan. 2015).

[14] R.I. Bahar et al. “Algebraic decision diagrams and their applications”. In: *Proceedings of 1993 International Conference on Computer Aided Design (ICCAD)*. 1993, pp. 188–191. DOI: 10.1109/ICCAD.1993.580054.

[15] C. Baier, M. Grosser, and F. Ciesinski. “Partial order reduction for probabilistic systems”. In: *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.* 2004, pp. 230–239. DOI: 10.1109/QEST.2004.1348037.

[16] Christel Baier, Marcus Größer, and Frank Ciesinski. “Partial order reduction for probabilistic systems”. In: *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.* (2004), pp. 230–239. URL: https://api.semanticscholar.org/CorpusID:196683.

[17] Davide Basile, Alessandro Fantechi, and Irene Rosadi. “Formal Analysis of the UNISIG Safety Application Intermediate Sub-layer”. In: *Formal Methods for Industrial Critical Systems*. Ed. by Alberto Lluch Lafuente and Anastasia Mavridou. Cham: Springer International Publishing, 2021, pp. 174–190. ISBN: 978-3-030-85248-1.

[18] Noor Cholis Basjaruddin, Didin Saefudin, and Nela Andriani. “Hardware Simulation of Camera-Based Adaptive Cruise Control Using Fuzzy Logic Control”. In: *Autom. Control Comput. Sci.* 55.6 (Nov. 2021), pp. 501–509. ISSN: 0146-4116. DOI: 10.3103/S0146411621060031. URL: https://doi.org/10.3103/S0146411621060031.

[19] Andreas Bauer, M. Leucker, and Christian Schallhart. “Monitoring of Real-Time Properties”. In: *Foundations of Software Technology and Theoretical Computer Science*. 2006. URL: https://api.semanticscholar.org/CorpusID:2852777.

[20] Gerd Behrmann et al. “UPPAAL 4.0”. In: *Third International Conference on the Quantitative Evaluation of Systems - (QEST’06)* (2006), pp. 125–126.

[21] Fatma Benkhelifa et al. “Recycling Cellular Energy for Self-Sustainable IoT Networks: A Spatiotemporal Study”. In: *IEEE Transactions on Wireless Communications* (Jan. 2020). DOI: 10.1109/TWC.2020.2967697.

[22] Laksh Bhatia et al. “Control Communication Co-Design for Wide Area Cyber-Physical Systems”. In: *ACM Trans. Cyber Phys. Syst.* 5.2 (2021), 18:1–18:27. DOI: 10.1145/3418528. URL: https://doi.org/10.1145/3418528.

[23] Armin Biere, Alessandro Cimatti, et al. “Symbolic Model Checking without BDDs”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by W. Rance Cleaveland. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 193–207. ISBN: 978-3-540-49059-3.

[24] Armin Biere, Edmund Clarke, et al. “Verifying Safety Properties of a PowerPC- Microprocessor Using Symbolic Model Checking without BDDs”. In: *Computer Aided Verification*. Ed. by Nicolas Halbwachs and Doron Peled. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 60–71. ISBN: 978-3-540-48683-1.

[25] Rafael H. Bordini et al. “State-Space Reduction Techniques in Agent Verification”. In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 2*. AAMAS ’04. New York, New York: IEEE Computer Society, 2004, pp. 896–903. ISBN: 1581138644.

[26] Rafael Heitor Bordini et al. “Verifying Multi-agent Programs by Model Checking”. In: *Autonomous Agents and Multi-Agent Systems* 12 (2006), pp. 239–256.

[27] D. Bosnacki, D.R. Dams, and L. Holenderski. “Symmetric Spin”. English. In: *SPIN model checking and software verification : 7th international SPIN workshop, Stanford CA, USA, August 30-September 1, 2000 : proceedings*. Ed. by K. Havelund, J. Penix, and W. Visser. Lecture Notes in Computer Science. Germany: Springer, 2000, pp. 1–19. ISBN: 3-540-41030-9. DOI: 10.1007/10722468_1.

[28] Amira Boulmaiz et al. “Chapter 9 - The use of WSN (wireless sensor network) in the surveillance of endangered bird species”. In: *Advances in Ubiquitous Computing*. Ed. by Amy Neustein. Advances in ubiquitous sensing applications for healthcare. Academic Press, 2020, pp. 261–306. DOI: https://doi.org/10.1016/B978-0-12-816801-1.00009-8. URL: https://www.sciencedirect.com/science/article/pii/B9780128168011000098.

[29] Bryant. “Graph-Based Algorithms for Boolean Function Manipulation”. In: *IEEE Transactions on Computers* C-35.8 (1986), pp. 677–691. DOI: 10.1109/TC.1986.1676819.

[30] Carlos E. Budde et al. “On Correctness, Precision, and Performance in Quantitative Verification: QComp 2020 Competition Report”. In: Rhodes, Greece: Springer-Verlag, 2020, pp. 216–241. ISBN: 978-3-030-83722-8. DOI: 10.1007/978-3-030-83723-5_15. URL: https://doi.org/10.1007/978-3-030-83723-5_15.

[31] J.R. Burch, E.M. Clarke, D.E. Long, et al. “Symbolic model checking for sequential circuit verification”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 13.4 (1994), pp. 401–424. DOI: 10.1109/43.275352.

[32] J.R. Burch, E.M. Clarke, K.L. McMillan, et al. “Symbolic model checking: 1020 States and beyond”. In: *Information and Computation* 98.2 (1992), pp. 142–170. ISSN: 0890-5401. DOI: https://doi.org/10.1016/0890-5401(92)90017-A. URL: https://www.sciencedirect.com/science/article/pii/089054019290017A.

[33] C. Cachin, K. Kursawe, and V. Shoup. “Random oracles in Constantinople: Practical asynchronous Byzantine agreement using cryptography (extended abstract)”. In: *Proc. Symposium on Principles of Distributed Computing*. 2000, pp. 123–132.

[34] M. Calder and A. Miller. *Five ways to use induction and symmetry in the verification of networks of processes by model-checking*. 2002. URL: http://eprints.gla.ac.uk/78678/.

[35] Muffy Calder, Simon Dobson, et al. “Making Sense of the World: Framing Models for Trustworthy Sensor-Driven Systems”. In: *Computers* 7.4 (2018). ISSN: 2073-431X. DOI: 10.3390/computers7040062. URL: https://www.mdpi.com/2073-431X/7/4/62.

[36] Muffy Calder and Alice Miller. “Automatic verification of any number of concurrent, communicating processes”. In: Feb. 2002, pp. 227–230. ISBN: 0-7695-1736-6. DOI: 10.1109/ASE.2002.1115017.

[37] Georgiana Caltais and Christian Schilling, eds. *Model Checking Software - 29th International Symposium, SPIN 2023, Paris, France, April 26-27, 2023, Proceedings*. Vol. 13872. Lecture Notes in Computer Science. Springer, 2023. ISBN: 978-3-031-32156-6. DOI: 10.1007/978-3-031-32157-3. URL: https://doi.org/10.1007/978-3-031-32157-3.

[38] J. Capetanakis. “Tree algorithms for packet broadcast channels”. In: *IEEE Transactions on Information Theory* 25.5 (1979), pp. 505–515. DOI: 10.1109/TIT.1979.1056093.

[39] Alessandro Cimatti et al. “NUSMV: a new symbolic model checker”. In: *STTT* 2 (Mar. 2000), pp. 410–425. DOI: 10.1007/s100090050046.

[40] E. M. Clarke, T. Filkorn, and S. Jha. “Exploiting symmetry in temporal logic model checking”. In: *Computer Aided Verification*. Ed. by Costas Courcoubetis. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 450–462. ISBN: 978-3-540-47787-7.

[41] E.M. Clarke, O. Grumberg, D. Kroening, et al. *Model Checking, second edition*. Cyber Physical Systems Series. MIT Press, 2018. ISBN: 9780262349451. URL: https://books.google.co.uk/books?id=qJl8DwAAQBAJ.

[42] Edmund Clarke, Armin Biere, et al. “Bounded Model Checking Using Satisfiability Solving”. In: *Form. Methods Syst. Des.* 19.1 (July 2001), pp. 7–34. ISSN: 0925-9856. DOI: 10.1023/A:1011276507260. URL: https://doi.org/10.1023/A:1011276507260.

[43] Edmund M. Clarke and E. Allen Emerson. “Design and synthesis of synchronization skeletons using branching time temporal logic”. In: *Logics of Programs*. Ed. by Dexter Kozen. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 52–71. ISBN: 978-3-540-39047-3.

[44] Edmund M. Clarke, Orna Grumberg, Somesh Jha, et al. “Counterexample-guided abstraction refinement for symbolic model checking”. In: *J. ACM* 50 (2003), pp. 752–794.

[45] Edmund M. Clarke, William Klieber, et al. “Model Checking and the State Explosion Problem”. In: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–30. ISBN: 978-3-642-35746-6. DOI: 10.1007/978-3-642-35746-6_1. URL: https://doi.org/10.1007/978-3-642-35746-6_1.

[46] Edmund M. Clarke and Bernd-Holger Schlingloff. “Chapter 24 - Model Checking”. In: *Handbook of Automated Reasoning*. Ed. by Alan Robinson and Andrei Voronkov. Handbook of Automated Reasoning. Amsterdam: North-Holland, 2001, pp. 1635–1790. ISBN: 978-0-444-50813-3. DOI: https://doi.org/10.1016/B978-044450813-3/50026-6. URL: https://www.sciencedirect.com/science/article/pii/B9780444508133500266.

[47] Edmund M. Clarke and Paolo Zuliani. “Statistical Model Checking for Cyber-Physical Systems”. In: *Automated Technology for Verification and Analysis*. Ed. by Tevfik Bultan and Pao-Ann Hsiung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 1–12. ISBN: 978-3-642-24372-1.

[48] Lora Alliance Technical Committee. *LoRaWAN 1.1 Specification*. URL: https://lora-alliance.org/sites/default/files/2018-07/lorawan1.0.3.pdf (visited on 07/07/2020).

[49] P. D’Argenio and Peter Niebert. “Partial order reduction on concurrent probabilistic programs”. In: *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.* (2004), pp. 240–249. URL: https://api.semanticscholar.org/CorpusID:15195776.

[50] Samar Dajani-Brown et al. “Formal Modeling and Analysis of an Avionics Triplex Sensor Voter”. In: *Proceedings of the 10th International Conference on Model Checking Software*. SPIN’03. Portland, OR, USA: Springer-Verlag, 2003, pp. 34–48. ISBN: 3540401172.

[51] Antônio Vicente Lourenço Dâmaso, Nelson Souto Rosa, and Paulo Romero Martins Maciel. "Using Coloured Petri Nets for Evaluating the Power Consumption of Wireless Sensor Networks". In: *International Journal of Distributed Sensor Networks* 10 (2014).

[52] Ornela Dardha. "Background on Session Types". In: *Type Systems for Distributed Programs: Components and Sessions*. Paris: Atlantis Press, 2016, pp. 61–71. ISBN: 978-94-6239-204-5. DOI: 10.2991/978-94-6239-204-5_5. URL: https://doi.org/10.2991/978-94-6239-204-5_5.

[53] Priyanka Darke, Sakshi Agrawal, and R. Venkatesh. "VeriAbs: A Tool for Scalable Verification by Abstraction (Competition Contribution)". In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Cham: Springer International Publishing, 2021, pp. 458–462. ISBN: 978-3-030-72013-1.

[54] Yuxin Deng and Davide Sangiorgi. "Ensuring termination by typability". In: *Information and Computation*. 2006.

[55] Maria Domenica Di Benedetto, Stefano Di Gennaro, and Alessandro D'Innocenzo. "Hybrid Systems and Verification by Abstraction". In: *Hybrid Dynamical Systems: Observation and Control*. Ed. by Mohamed Djemai and Michael Defoort. Cham: Springer International Publishing, 2015, pp. 1–25. ISBN: 978-3-319-10795-0. DOI: 10.1007/978-3-319-10795-0_1. URL: https://doi.org/10.1007/978-3-319-10795-0_1.

[56] S. Dolev, A. Israeli, and S. Moran. "Analyzing Expected Time by Scheduler-Luck Games". In: *IEEE Transactions on Software Engineering* 21.5 (1995), pp. 429–439.

[57] A. Donaldson and A. Miller. "Symmetry Reduction for Probabilistic Model Checking using Generic Representatives". In: *Proc. 4th Int. Symp. Automated Technology for Verification and Analysis (ATVA'06)*. Ed. by S. Graf and W. Zhang. Vol. 4218. Lecture Notes in Computer Science. Springer, 2006, pp. 9–23.

[58] A. Donaldson, A. Miller, and D. Parker. "Language-level Symmetry Reduction for Probabilistic Model Checking". In: *Proc. 6th International Conference on Quantitative Evaluation of Systems (QEST'09)*. IEEE Computer Society, 2009, pp. 289–298.

[59] Alastair Donaldson, Alice Miller, and David Parker. "Language-Level Symmetry Reduction for Probabilistic Model Checking". In: *QEST - 6th International Conference on the Quantitative Evaluation of Systems*. Oct. 2009, pp. 289–298. DOI: 10.1109/QEST.2009.21.

[60] Alastair F. Donaldson and Alice Miller. "Automatic Symmetry Detection for Model Checking Using Computational Group Theory". In: *World Congress on Formal Methods*. 2005.

[61] Alastair F. Donaldson and Alice Miller. "Automatic Symmetry Detection for Promela". In: *Journal of Automated Reasoning* 41 (2008), pp. 251–293. URL: https://api.semanticscholar.org/CorpusID:16298636.

[62] Alastair F. Donaldson and Alice Miller. "Evaluating a formal methods technique via student assessed exercises". In: *Formal Methods in the Teaching Lab: Examples, Cases, Assignments and Projects Enhancing Formal Methods Education. Workshop at the Formal Methods 2006 Symposium, Hamilton, Ontario, Canada*. 2006, pp. 93–98.

[63] Alastair F. Donaldson and Alice Miller. "Exact and Approximate Strategies for Symmetry Reduction in Model Checking". In: *World Congress on Formal Methods*. 2006.

[64] Alastair F. Donaldson and Alice Miller. "Symmetry reduction techniques for explicit-state model checking". In: *First International Symmetry Conference*. Edinburgh, UK, Jan. 2007, pp. 41–45.

[65] Alastair F. Donaldson, Alice Miller, and David Parker. "GRIP: Generic representatives in PRISM". In: *in Proceedings of the 4th International Conference on Quantitative Evaluation of Systems (QEST'07). IEEE Computer Society*, pp. 115–116.

[66] Jin Song Dong et al. "Specifying and Verifying Sensor Networks: An Experiment of Formal Methods". In: *IEEE International Conference on Formal Engineering Methods*. 2008.

[67] Jean-François Dufourd. "Pointer Program Derivation Using Coq: Graphs and Schorr-Waite Algorithm". In: *Formal Methods and Software Engineering*. Ed. by Stephan Merz and Jun Pang. Cham: Springer International Publishing, 2014, pp. 139–154. ISBN: 978-3-319-11737-9.

[68] Stephen Edwards et al. "Design of Embedded Systems: Formal Models, Validation, and Synthesis". In: *Readings in Hardware/Software Co-Design*. Ed. by Giovanni De Micheli, Rolf Ernst, and Wayne Wolf. Systems on Silicon. San Francisco: Morgan Kaufmann, 2002, pp. 86–107. DOI: https://doi.org/10.1016/B978-155860702-6/50009-0. URL: https://www.sciencedirect.com/science/article/pii/B9781558607026500090.

[69] E. Emerson and A. Sistla. "Symmetry and Model Checking". In: *Formal Methods in System Design* 9 (Aug. 1996), pp. 105–131. DOI: 10.1007/BF00625970.

[70] E. Allen Emerson and Thomas Wahl. "Dynamic Symmetry Reduction". In: *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. 2005.

[71] E. Allen Emerson and Thomas Wahl. "On Combining Symmetry Reduction and Symbolic Representation for Efficient Model Checking". In: *In Conference on Correct Hardware Design and Verification Methods (CHARME*. Springer, 2003, pp. 216–230.

[72] Shifeng Fang et al. “An integrated approach to snowmelt flood forecasting in water resource management”. In: *IEEE Transactions on Industrial Informatics* 10.1 (2014). Cited by: 101, pp. 548–558. DOI: 10.1109/TII.2013.2257807. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84890935327&doi=10.1109%2fTII.2013.2257807&partnerID=40&md5=0cc558e79e3ebac0d5a85113c327db41.

[73] Ansgar Fehnker et al. “A Process Algebra for Wireless Mesh Networks”. In: *Programming Languages and Systems*. Springer Berlin Heidelberg, 2012, pp. 295–315. DOI: 10.1007/978-3-642-28869-2_15. URL: https://doi.org/10.1007%2F978-3-642-28869-2_15.

[74] Jinglang Feng, Ron Noomen, et al. “Modeling and analysis of periodic orbits around a contact binary asteroid”. In: *Astrophysics and Space Science* 357.2 (2015). Cited by: 12; All Open Access, Green Open Access, Hybrid Gold Open Access. DOI: 10.1007/s10509-015-2353-0. URL: https://www.scopus.com/inward/record.uri?eid=2-s2.0-84929095035&doi=10.1007%2fs10509-015-2353-0&partnerID=40&md5=688a70a6a0a9f1c6d25ecef81cf7e292.

[75] Yuan Feng, Ernst Moritz Hahn, et al. “Model Checking ω-regular Properties for Quantum Markov Chains”. In: 2017.

[76] Jaroslav Fogel. “A Survey of Verification Techniques for Solving the State Explosion Problem”. In: *IFAC Proceedings Volumes* 33.13 (2000). IFAC Conference on Control Systems Design (CSD 2000), Bratislava, Slovak Republic, 18-20 June 2000, pp. 361–366. ISSN: 1474-6670. DOI: https://doi.org/10.1016/S1474-6670(17)37216-6. URL: https://www.sciencedirect.com/science/article/pii/S1474667017372166.

[77] *Formal Methods and Software Engineering: 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24–27, 2022, Proceedings*. Madrid, Spain: Springer-Verlag, 2022. ISBN: 978-3-031-17243-4.

[78] Jeffrey S. Foster, Michael W. Hicks, and William Pugh. “Improving Software Quality with Static Analysis”. In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE ’07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 83–84. ISBN: 9781595935953. DOI: 10.1145/1251535.1251549. URL: https://doi.org/10.1145/1251535.1251549.

[79] Gordon Fraser and José Miguel Rojas. “Software Testing”. In: *Handbook of Software Engineering*. Ed. by Sungdeok Cha, Richard N. Taylor, and Kyochul Kang. Cham: Springer International Publishing, 2019, pp. 123–192. ISBN: 978-3-030-00262-6. DOI:

10.1007/978-3-030-00262-6_4. URL: https://doi.org/10.1007/978-3-030-00262-6_4.

[80] M. Fruth. "Formal Methods for the Analysis of Wireless Network Protocols". PhD thesis. Oxford University, 2011.

[81] Chen Fu, Ernst Moritz Hahn, et al. "EPMC Gets Knowledge in Multi-agent Systems". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Bernd Finkbeiner and Thomas Wies. Cham: Springer International Publishing, 2022, pp. 93–107. ISBN: 978-3-030-94583-1.

[82] Chen Fu, Andrea Turrini, et al. "Model Checking Probabilistic Epistemic Logic for Probabilistic Multiagent Systems". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 4757–4763. DOI: 10.24963/ijcai.2018/661. URL: https://doi.org/10.24963/ijcai.2018/661.

[83] M. Fujita, P. C. McGeer, and J. C.-Y. Yang. "Multi-Terminal Binary Decision Diagrams: An Efficient DataStructure for Matrix Representation". In: 10.2–3 (Apr. 1997), pp. 149–169. ISSN: 0925-9856. DOI: 10.1023/A:1008647823331. URL: https://doi.org/10.1023/A:1008647823331.

[84] Etienne Gagnon and Laurie Hendren. "SableCC An Object-Oriented Compiler Framework". In: *Proceedings of TOOLS 1998* (Apr. 1998). DOI: 10.1109/TOOLS.1998.711009.

[85] Paul Gainer, Clare Dixon, and Ullrich Hustadt. "Probabilistic Model Checking of Ant-Based Positionless Swarming". In: *Towards Autonomous Robotic Systems*. Ed. by Lyuba Alboul, Dana Damian, and Jonathan M. Aitken. Cham: Springer International Publishing, 2016, pp. 127–138. ISBN: 978-3-319-40379-3.

[86] Paul Gainer, Sven Linker, et al. "Investigating Parametric Influence on Discrete Synchronisation Protocols Using Quantitative Model Checking". In: *Quantitative Evaluation of Systems*. Ed. by Nathalie Bertrand and Luca Bortolussi. Cham: Springer International Publishing, 2017, pp. 224–239. ISBN: 978-3-319-66335-7.

[87] Paul Gainer, Sven Linker, et al. *Multi-Scale Verification of Distributed Synchronisation*. Sept. 2018.

[88] S. Gay and A. Ravara. *Behavioural Types: from Theory to Tools*. River Publishers Series in Automation, Control and Robotics. River Publishers, 2017. ISBN: 9788793519824. URL: https://books.google.co.uk/books?id=6x0vDwAAQBAJ.

[89] Simon J. Gay and Malcolm Hole. "Subtyping for session types in the pi calculus". In: *Acta Informatica* 42 (2005), pp. 191–225.

[90] B. Gebremichael-Tesfagiorgis, Frits W. Vaandrager, and M. Zhang. "Analysis of a Protocol for Dynamic Configuration of IPv4 Link Local Addresses Using Uppaal". In: *CTIT technical report series* (2006).

[91] L. Georgiadis and P. Papantoni-Kazakos. "A 0.487 throughput limited sensing algorithm". In: *IEEE Transactions on Information Theory* 33.2 (1987), pp. 233–237. DOI: 10.1109/TIT.1987.1057278.

[92] P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems*. Vol. 1032. Berlin, 1996.

[93] Patrice Godefroid and Didier Pirottin. "Refining dependencies improves partial-order verification methods (extended abstract)". In: *Computer Aided Verification*. Ed. by Costas Courcoubetis. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 438–449. ISBN: 978-3-540-47787-7.

[94] Anjana Gosain and Ganga Sharma. "A Survey of Dynamic Program Analysis Techniques and Tools". In: *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014*. Ed. by Suresh Chandra Satapathy et al. Cham: Springer International Publishing, 2015, pp. 113–122. ISBN: 978-3-319-11933-5.

[95] D. Graham, M. Calder, and A. Miller. "An inductive technique for parameterised model checking of degenerative distributed randomised protocols". In: *Electronic Notes in Theoretical Computer Science* 250.1 (2009), pp. 87–103. DOI: 10.1016/j.entcs.2009.08.007. URL: http://eprints.gla.ac.uk/39289/.

[96] *GRIP source code*. https://github.com/afd/symmetrytools/. Accessed: 2023-03-27.

[97] *GRIP website*. www.prismmodelchecker.org/grip. Accessed: 2023-02-27.

[98] Ian Grout. "CHAPTER 2 - Electronic Systems Design". In: *Digital Systems Design with FPGAs and CPLDs*. Ed. by Ian Grout. Burlington: Newnes, 2008, pp. 43–121. ISBN: 978-0-7506-8397-5. DOI: https://doi.org/10.1016/B978-0-7506-8397-5.00002-7. URL: https://www.sciencedirect.com/science/article/pii/B9780750683975000027.

[99] "Relaxed visibility enhances partial order reduction". In: *Computer Aided Verification - 9th International Conference, CAV 1997, Proceedings*. Ed. by Orna Grumberg. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer Verlag, 1997, pp. 328–339. ISBN: 3540631666. DOI: 10.1007/3-540-63166-6_33.

[100] Orna Grumberg and Helmut Veith. *25 Years of Model Checking - History, Achievements, Perspectives*. Vol. 5000. Jan. 2008. ISBN: 978-3-540-69849-4. DOI: 10.1007/978-3-540-69850-0.

[101] Safa Guellouz et al. "Designing Efficient Reconfigurable Control Systems Using IEC61499 and Symbolic Model Checking". In: *IEEE Transactions on Automation Science and Engineering* 16.3 (2019), pp. 1110–1124. DOI: 10.1109/TASE.2018.2868897.

[102] Susmita Guha, Akash Nag, and Rahul Karmakar. "Formal Verification of Safety-Critical Systems: A Case-Study in Airbag System Design". In: *Intelligent Systems Design and Applications*. Ed. by Ajith Abraham et al. Cham: Springer International Publishing, 2021, pp. 107–116. ISBN: 978-3-030-71187-0.

[103] Ernst Hahn, Arnd Hartmanns, et al. "The 2019 Comparison of Tools for the Analysis of Quantitative Formal Models: (QComp 2019 Competition Report)". In: Apr. 2019, pp. 69–92. ISBN: 978-1-4939-9100-6. DOI: 10.1007/978-3-030-17502-3_5.

[104] Ernst Moritz Hahn, Guangyuan Li, et al. "Lazy Determinisation for Quantitative Model Checking". In: *ArXiv* abs/1311.2928 (2013).

[105] Gertjan P. Halkes and Koen Langendoen. "Energy-Efficient Medium Access Control". In: *Embedded Systems Handbook*. 2005.

[106] Sylvain Hallé et al. "A Formal Validation Model for the Netconf Protocol". In: *Utility Computing*. Ed. by Akhil Sahai and Felix Wu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 147–158. ISBN: 978-3-540-30184-4.

[107] Hans Hansson and Bengt Jonsson. "A Logic for Reasoning about Time and Reliability". In: *Form. Asp. Comput.* 6.5 (Sept. 1994), pp. 512–535. ISSN: 0934-5043. DOI: 10.1007/BF01211866. URL: https://doi.org/10.1007/BF01211866.

[108] Justin E. Harlow and Franc Brglez. "Design of experiments and evaluation of BDD ordering heuristics". In: *International Journal on Software Tools for Technology Transfer* 3 (2001), pp. 193–206.

[109] J. Heath et al. "Probabilistic model checking of complex biological pathways". In: *Proc. Computational Methods in Systems Biology (CMSB'06)*. Ed. by C. Priami. Vol. 4210. Lecture Notes in Bioinformatics. Springer Verlag, 2006, pp. 32–47.

[110] J. Heath et al. "Probabilistic model checking of complex biological pathways". In: *Theoretical Computer Science* 319.3 (2008), pp. 239–257.

[111] Martijn Hendriks et al. "Adding Symmetry Reduction to Uppaal". In: *International Conference on Formal Modeling and Analysis of Timed Systems*. 2003.

[112] Christian Hensel et al. "The Probabilistic Model Checker Storm". In: *CoRR* abs/2002.07080 (2020). arXiv: 2002.07080. URL: https://arxiv.org/abs/2002.07080.

[113] Ruth Hoffmann et al. "Autonomous Agent Behaviour Modelled in PRISM – A Case Study". In: *Model Checking Software*. Ed. by Dragan Bošnački and Anton Wijs. Cham: Springer International Publishing, 2016, pp. 104–110. ISBN: 978-3-319-32582-8.

[114] Gerard Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. 1st. Addison-Wesley Professional, 2011.

[115] Gerard J. Holzmann and Margaret H. Smith. "Automating software feature verification". In: *Bell Labs Technical Journal* 5 (2000), pp. 72–87.

[116] Xiaowei Huang, Marta Kwiatkowska, et al. "Safety Verification of Deep Neural Networks". In: *CoRR* abs/1610.06940 (2016). arXiv: 1610.06940.

[117] Xiaoxia Huang and Yuguang Fang. "Multiconstrained QoS multipath routing in wireless sensor networks". In: *Wireless Networks* 14 (2008), pp. 465–478.

[118] M. Huisman, C. Păsăreanu, and N. Zhan. *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings*. Lecture Notes in Computer Science. Springer International Publishing, 2021. ISBN: 9783030908706. URL: https://books.google.co.uk/books?id=l4ZNEAAAQBAJ.

[119] Pierre A. Humblet. "On the Throughput of Channel Access Algorithms with Limited Sensing". In: *IEEE Trans. Commun.* 34 (1986), pp. 345–347.

[120] "IEEE Standard for Information technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements Part 3: Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications". In: *IEEE Std 802.3-2002 (Revision of IEEE Std 802.3, 2000 edn)* (2002), pp. 1–1550. DOI: 10.1109/IEEESTD.2002.93570.

[121] Daniel Jackson. "Alloy: A Lightweight Object Modelling Notation". In: *ACM Trans. Softw. Eng. Methodol.* 11.2 (Apr. 2002), pp. 256–290. ISSN: 1049-331X. DOI: 10.1145/505145.505149.

[122] Márk Jelasity et al. "Gossip-Based Peer Sampling". In: *ACM Trans. Comput. Syst.* 25.3 (Aug. 2007), 8–es. ISSN: 0734-2071. DOI: 10.1145/1275517.1275520. URL: https://doi.org/10.1145/1275517.1275520.

[123] Henrik Jensen, Kim Larsen, and Arne Skou. "Modelling and Analysis of a Collision Avoidance Protocol using SPIN and UPPAAL". In: *BRICS Report Series* 3 (Jan. 2002). DOI: 10.7146/brics.v3i24.20005.

[124] Zill-E-Huma Kamal and Mohammad Salahuddin. "Introduction to Wireless Sensor Networks". In: Jan. 2015, pp. 3–32. ISBN: 978-1-4939-2468-4. DOI: 10.1007/978-1-4939-2468-4_1.

[125] JN Al-Karaki and Ahmed Kamal. "Routing Techniques in Wireless Sensor Networks: A Survey". In: *Wireless Communications, IEEE* 11 (Jan. 2005), pp. 6–28. DOI: 10.1109/MWC.2004.1368893.

[126] Pim Kars. "The application of Promela and Spin in the BOS project". In: *The Spin Verification System*. 1996.

[127] Joost-Pieter Katoen et al. "The Ins and Outs of the Probabilistic Model Checker MRMC". In: *2009 Sixth International Conference on the Quantitative Evaluation of Systems*. 2009, pp. 167–176. DOI: 10.1109/QEST.2009.11.

[128] Mark Kattenbelt et al. "Abstraction Refinement for Probabilistic Software". In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Neil D. Jones and Markus Müller-Olm. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 182–197. ISBN: 978-3-540-93900-9.

[129] Shmuel Katz and Doron Peled. "An efficient verification method for parallel and distributed programs". In: *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*. Ed. by J. W. de Bakker, W. -P. de Roever, and G. Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1989, pp. 489–507. ISBN: 978-3-540-46147-0.

[130] Shmuel Katz and Doron Peled. "Defining conditional independence using collapses". In: *Theoretical Computer Science* 101.2 (1992), pp. 337–359. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(92)90054-J. URL: https://www.sciencedirect.com/science/article/pii/030439759290054J.

[131] Lucia Keleadile Ketshabetswe et al. "Communication protocols for wireless sensor networks: A survey and comparison". In: *Heliyon* 5.5 (2019), e01591. ISSN: 2405-8440. DOI: https://doi.org/10.1016/j.heliyon.2019.e01591. URL: https://www.sciencedirect.com/science/article/pii/S2405844018340192.

[132] Shahid Khan et al. "Modelling and Analysis of Fire Sprinklers by Verifying Dynamic Fault Trees". In: *2021 10th Latin-American Symposium on Dependable Computing (LADC)*. 2021, pp. 1–10. DOI: 10.1109/LADC53747.2021.9672579.

[133] Hokeun Kim et al. “A Toolkit for Construction of Authorization Service Infrastructure for the Internet of Things”. In: *Proceedings of the 2nd ACM/IEEE International Conference on Internet-of-Things Design and Implementation (IoTDI)*. Winner of the Best Paper Award, presented during CPS Week, 2017, Pittsburgh, PA. Apr. 2017. URL: http://chess.eecs.berkeley.edu/pubs/1187.html.

[134] Ryan Kirwan et al. “Formal Modeling of Robot Behavior with Learning”. In: *Neural Computation* 25.11 (Nov. 2013), pp. 2976–3019. ISSN: 0899-7667. DOI: 10.1162/NECO_a_00493. eprint: https://direct.mit.edu/neco/article-pdf/25/11/2976/901360/neco_a_00493.pdf. URL: https://doi.org/10.1162/NECO%5C_a%5C_00493.

[135] Joseph Migga Kizza. “Software Issues: Risks and Liabilities”. In: *Ethical and Secure Computing: A Concise Module*. Cham: Springer International Publishing, 2019, pp. 149–176. ISBN: 978-3-030-03937-0. DOI: 10.1007/978-3-030-03937-0_7. URL: https://doi.org/10.1007/978-3-030-03937-0_7.

[136] Jan Kleissl and Yuvraj Agarwal. “Cyber-physical energy systems: Focus on smart buildings”. In: *Design Automation Conference*. 2010, pp. 749–754. DOI: 10.1145/1837274.1837464.

[137] Vasileios Klimis et al. “Taking Back Control in an Intermediate Representation for GPU Computing”. In: *Proceedings of the ACM on Programming Languages* 7 (2023), pp. 1740–1769.

[138] Naoki Kobayashi. “A New Type System for Deadlock-Free Processes”. In: *CONCUR 2006 – Concurrency Theory*. Ed. by Christel Baier and Holger Hermanns. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 233–247. ISBN: 978-3-540-37377-3.

[139] Savas Konur, Clare Dixon, and Michael Fisher. “Analysing Robot Swarm Behaviour via Probabilistic Model Checking”. In: *Robotics and Autonomous Systems* 60 (Feb. 2012), pp. 199–213. DOI: 10.1016/j.robot.2011.10.005.

[140] Thomas Kropf. “Introduction to Formal Hardware Verification: Methods and Tools for Designing Correct Circuits and Systems”. In: 1999.

[141] Marcin Kubica, Adam Opara, and Dariusz Kania. “Ordering Variables in BDD Diagrams”. In: *Technology Mapping for LUT-Based FPGA*. Cham: Springer International Publishing, 2021, pp. 65–70. ISBN: 978-3-030-60488-2. DOI: 10.1007/978-3-030-60488-2_6. URL: https://doi.org/10.1007/978-3-030-60488-2_6.

[142] Tomas Kulik et al. “A Framework for Threat-Driven Cyber Security Verification of IoT Systems”. In: *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 2018, pp. 89–97. DOI: 10.1109/ICSTW.2018.00033.

[143] M. Kwiatkowska, G. Norman, and D. Parker. “PRISM 4.0: Verification of Probabilistic Real-Time Systems”. In: *CAV 2011*, pp. 585–591.

[144] M. Kwiatkowska, G. Norman, and D. Parker. “Symmetry Reduction for Probabilistic Model Checking”. In: *Proc. 18th International Conference on Computer Aided Verification (CAV’06)*. Ed. by T. Ball and R. Jones. Vol. 4114. LNCS. Springer, 2006, pp. 234–248.

[145] M. Kwiatkowska, G. Norman, and R. Segala. “Automated Verification of a Randomized Distributed Consensus Protocol Using Cadence SMV and PRISM”. In: *Proc. 13th International Conference on Computer Aided Verification (CAV’01)*. Ed. by G. Berry, H. Comon, and A. Finkel. Vol. 2102. LNCS. Springer, 2001, pp. 194–206.

[146] M. Kwiatkowska, G. Norman, and J. Sproston. “Probabilistic Model Checking of the IEEE 802.11 Wireless Local Area Network Protocol”. In: *Proc. 2nd Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV’02)*. Ed. by H. Hermanns and R. Segala. Vol. 2399. LNCS. Springer, 2002, pp. 169–187.

[147] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. “Symbolic Model Checking for Probabilistic Timed Automata”. In: *Proc. Joint Conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant Systems (FORMATS/FTRTFT’04)*. Ed. by Y. Lakhnech and S. Yovine. Vol. 3253. LNCS. Springer, 2004, pp. 293–308.

[148] M. Kwiatkowska, G. Norman, J. Sproston, and F. Wang. “Symbolic Model Checking for Probabilistic Timed Automata”. In: *Information and Computation* 205.7 (2007), pp. 1027–1077.

[149] Marta Kwiatkowska, Gethin Norman, and David Parker. “Probabilistic Model Checking: Advances and Applications”. In: *Formal System Verification: State-of the-Art and Future Trends*. Ed. by Rolf Drechsler. Cham: Springer International Publishing, 2018, pp. 73–121. ISBN: 978-3-319-57685-5. DOI: 10.1007/978-3-319-57685-5_3. URL: https://doi.org/10.1007/978-3-319-57685-5_3.

[150] Ivan Lanese and Davide Sangiorgi. “An operational semantics for a calculus for wireless systems”. In: *Theor. Comput. Sci.* 411 (2010), pp. 1928–1948.

[151] Koen Langendoen, Aline Baggio, and Otto Visser. "Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture". In: *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium* (2006), 8–pp.

[152] Paola Lecca and Corrado Priami. "Cell Cycle Control in Eukaryotes: A BioSpi model". In: *BioConcur@CONCUR*. 2007. URL: https://api.semanticscholar.org/CorpusID:52809002.

[153] Benjamin C. Lee. "Hardware Simulation". In: *Datacenter Design and Management: A Computer Architect's Perspective*. Cham: Springer International Publishing, 2016, pp. 55–78. ISBN: 978-3-031-01752-0. DOI: 10.1007/978-3-031-01752-0_5. URL: https://doi.org/10.1007/978-3-031-01752-0_5.

[154] Choong Y. Lee. "Representation of switching circuits by binary-decision programs". In: *Bell System Technical Journal* 38 (1959), pp. 985–999.

[155] Axel Legay, Benoît Delahaye, and Saddek Bensalem. "Statistical Model Checking: An Overview". In: *Runtime Verification*. Ed. by Howard Barringer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 122–135. ISBN: 978-3-642-16612-9.

[156] Axel Legay, Anna Lukina, et al. "Statistical Model Checking". In: *Computing and Software Science: State of the Art and Perspectives*. Ed. by Bernhard Steffen and Gerhard Woeginger. Cham: Springer International Publishing, 2019, pp. 478–504. ISBN: 978-3-319-91908-9. DOI: 10.1007/978-3-319-91908-9_23. URL: https://doi.org/10.1007/978-3-319-91908-9_23.

[157] D. Lehmann and M. Rabin. "On the Advantage of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem (Extended Abstract)". In: *Proc. 8th Annual ACM Symposium on Principles of Programming Languages (POPL'81)*. 1981, pp. 133–138.

[158] Michael Leuschel. "The High Road to Formal Validation:" in: *Abstract State Machines, B and Z*. Ed. by Egon Börger et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 4–23. ISBN: 978-3-540-87603-8.

[159] N.G. Leveson and C.S. Turner. "An investigation of the Therac-25 accidents". In: *Computer* 26.7 (1993), pp. 18–41. DOI: 10.1109/MC.1993.274940.

[160] Xu Li et al. "Smart community: an internet of things application". In: *IEEE Communications Magazine* 49.11 (2011), pp. 68–75. DOI: 10.1109/MCOM.2011.6069711.

[161] David Liben-Nowell, Hari Balakrishnan, and David Karger. "Analysis of the Evolution of Peer-to-Peer Systems". In: *Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing*. PODC '02. Monterey, California: Association for Computing Machinery, 2002, pp. 233–242. ISBN: 1581134851. DOI: 10.1145/571825.571863. URL: https://doi.org/10.1145/571825.571863.

[162] Chenyang Lu, Abusayeed Saifullah, et al. "Real-Time Wireless Sensor-Actuator Networks for Industrial Cyber-Physical Systems". In: *Proc. IEEE* 104.5 (2016), pp. 1013–1024. DOI: 10.1109/JPROC.2015.2497161. URL: https://doi.org/10.1109/JPROC.2015.2497161.

[163] Yu Lu, Alice Miller, et al. "Availability Analysis of Satellite Positioning Systems for Aviation Using the PRISM Model Checker". In: *2014 IEEE 17th International Conference on Computational Science and Engineering*. 2014, pp. 704–713. DOI: 10.1109/CSE.2014.148.

[164] Yu Lu, Zhaoguang Peng, et al. "How reliable is satellite navigation for aviation? Checking availability properties with probabilistic verification". In: *Reliability Engineering & System Safety* 144 (2015), pp. 95–116. ISSN: 0951-8320. DOI: https://doi.org/10.1016/j.ress.2015.07.020. URL: https://www.sciencedirect.com/science/article/pii/S0951832015002252.

[165] José A. Mateo et al. "Probabilistic Model Checking: One Step Forward in Wireless Sensor Networks Simulation". In: *International Journal of Distributed Sensor Networks* 11.5 (2015), p. 285396. DOI: 10.1155/2015/285396. eprint: https://doi.org/10.1155/2015/285396. URL: https://doi.org/10.1155/2015/285396.

[166] José A. Mateo et al. "Probabilistic Model Checking: One Step Forward in Wireless Sensor Networks Simulation". In: *International Journal of Distributed Sensor Networks* 11.5 (2015), p. 285396. DOI: 10.1155/2015/285396. eprint: https://doi.org/10.1155/2015/285396. URL: https://doi.org/10.1155/2015/285396.

[167] K.L. McMillan. *Symbolic Model Checking*. Boston, U.S.A., 1993.

[168] KL McMillan and James Schwalbe. "Formal verification of the gigamax cache consistency protocol". In: *Proceedings of the International Symposium on Shared Memory Multiprocessing*. 1992, pp. 111–134.

[169] Francesco Mercaldo, Fabio Martinelli, and Antonella Santone. "Real-Time SCADA Attack Detection by Means of Formal Methods". In: *2019 IEEE 28th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*. 2019, pp. 231–236. DOI: 10.1109/WETICE.2019.00057.

[170] A. Miller and M. Calder. *An application of abstraction and induction techniques to degenerating systems of processes*. 2003. URL: http://eprints.gla.ac.uk/78674/.

[171] A. Miller and A. Donaldson. *Property preservation in quotient structures*. Tech. rep. 2008.

[172] A. Miller, A. Donaldson, and M. Calder. “Symmetry in temporal logic model checking”. In: *ACM Computing Surveys* 38.3 (Sept. 2006). DOI: 10.1145/1132960.1132962. URL: http://eprints.gla.ac.uk/3197/.

[173] Nazeeruddin Mohammad et al. “Design and modeling of energy efficient wsn architecture for tactical applications”. In: *2017 Military Communications and Information Systems Conference (MilCIS)*. IEEE. 2017, pp. 1–6.

[174] Abhijit Mohanta and Anoop Saldanha. “Static Analysis”. In: *Malware Analysis and Detection Engineering: A Comprehensive Approach to Detect and Analyze Modern Malware*. Berkeley, CA: Apress, 2020, pp. 377–402. ISBN: 978-1-4842-6193-4. DOI: 10.1007/978-1-4842-6193-4_12. URL: https://doi.org/10.1007/978-1-4842-6193-4_12.

[175] Mujahid Mohsin et al. “IoTRiskAnalyzer: A Probabilistic Model Checking Based Framework for Formal Risk Analytics of the Internet of Things”. In: *IEEE Access* 5 (2017), pp. 5494–5505.

[176] Alexandre Mouradian and Isabelle Augé-Blum. “Formal verification of real-time wireless sensor networks protocols with realistic radio links”. In: *International Conference on Real-Time and Network Systems*. 2013.

[177] Khoa Ngo, Trong Huynh, and De Huynh. “Simulation Wireless Sensor Networks in Castalia”. In: Feb. 2018, pp. 39–44. DOI: 10.1145/3193063.3193066.

[178] Gerard O’Regan. “Software Testing”. In: *Concise Guide to Software Engineering: From Fundamentals to Application Methods*. Cham: Springer International Publishing, 2022, pp. 137–153. ISBN: 978-3-031-07816-3. DOI: 10.1007/978-3-031-07816-3_8. URL: https://doi.org/10.1007/978-3-031-07816-3_8.

[179] Gerard O’Regan. “Verification of Safety-Critical Systems”. In: *Concise Guide to Software Testing*. Cham: Springer International Publishing, 2019, pp. 235–250. ISBN: 978-3-030-28494-7. DOI: 10.1007/978-3-030-28494-7_13. URL: https://doi.org/10.1007/978-3-030-28494-7_13.

[180] Iulian Ober, Susanne Graf, and David Lesens. “Modeling and Validation of a Software Architecture for the Ariane-5 Launcher”. In: *Formal Methods for Open Object-Based Distributed Systems*. Ed. by Roberto Gorrieri and Heike Wehrheim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 48–62. ISBN: 978-3-540-34895-5.

[181] Pangun Park et al. “Wireless Network Design for Control Systems: A Survey”. In: *IEEE Communications Surveys & Tutorials* 20 (2017), pp. 978–1013.

[182] M. Paterakis. “A Limited Sensing Random Access Algorithm With Binary Success-failure Feedback”. In: *Twenty-Second Asilomar Conference on Signals, Systems and Computers*. Vol. 1. 1988, pp. 97–101. DOI: 10.1109/ACSSC.1988.753961.

[183] M. Paterakis, Leonidas Georgiadis, and P. Papantoni-Kazakos. “A full sensing window Random-Access algorithm for messages with strict delay constraints”. In: *Algorithmica* 4 (Jan. 1989), pp. 313–328. DOI: 10.1007/BF01553894.

[184] M. Paterakis and P. Papantoni-Kazakos. “A simple window random access algorithm with advantageous properties”. In: *IEEE Transactions on Information Theory* 35.5 (1989), pp. 1124–1130. DOI: 10.1109/18.42234.

[185] Doron Peled. “All from one, one for all: on model checking using representatives”. In: *Computer Aided Verification*. Ed. by Costas Courcoubetis. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 409–423. ISBN: 978-3-540-47787-7.

[186] Doron Peled. “Partial-Order Reduction”. In: *Handbook of Model Checking*. Ed. by Edmund M. Clarke et al. Cham: Springer International Publishing, 2018, pp. 173–190. ISBN: 978-3-319-10575-8. DOI: 10.1007/978-3-319-10575-8_6. URL: https://doi.org/10.1007/978-3-319-10575-8_6.

[187] Zhaoguang Peng et al. “Formal Specification and Quantitative Analysis of a Constellation of Navigation Satellites”. In: *Quality and Reliability Engineering International* 32.2 (2016), pp. 345–361. DOI: https://doi.org/10.1002/qre.1754. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/qre.1754. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/qre.1754.

[188] Radia J. Perlman. “An algorithm for distributed computation of a spanningtree in an extended LAN”. In: *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. 1985.

[189] Sophia Petridou, Stylianos Basagiannis, and Manos Roumeliotis. “Survivability analysis using probabilistic model checking: A study on wireless sensor networks”. In: *IEEE systems journal* 7.1 (2012), pp. 4–12.

[190] Alberto Pettorossi and Maurizio Proietti. “Program Derivation = Rules + Strategies”. In: *Computational Logic: Logic Programming and Beyond: Essays in Honour of Robert A. Kowalski Part I*. Ed. by Antonis C. Kakas and Fariba Sadri. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 273–309. ISBN: 978-3-540-45628-5. DOI: 10.1007/3-540-45628-7_12. URL: https://doi.org/10.1007/3-540-45628-7_12.

[191] A. Pnueli and L. Zuck. “Verification of Multiprocess Probabilistic Protocols”. In: *Distributed Computing* 1.1 (1986), pp. 53–72.

[192] Vidyasagar Potdar, Atif Sharif, and Elizabeth Chang. “Wireless Sensor Networks: A Survey”. In: *2009 International Conference on Advanced Information Networking and Applications Workshops*. 2009, pp. 636–641. DOI: 10.1109/WAINA.2009.192.

[193] *Prism - Case Studies*. https://www.prismmodelchecker.org/casestudies/index.php. Accessed: 2023-05-19.

[194] Tim Quatmann, Christian Dehnert, et al. *Parameter Synthesis for Markov Models: Faster Than Ever*. 2016. arXiv: 1602.05113 [cs.LO].

[195] Tim Quatmann, Sebastian Junges, and Joost-Pieter Katoen. "Markov automata with multiple objectives". In: *Formal Methods in System Design* 60 (Mar. 2021). DOI: 10.1007/s10703-021-00364-6.

[196] J. P. Queille and J. Sifakis. "Specification and verification of concurrent systems in CESAR". In: *International Symposium on Programming*. Ed. by Mariangiola Dezani-Ciancaglini and Ugo Montanari. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 337–351. ISBN: 978-3-540-39184-5.

[197] M. Rabin. "N-Process Mutual Exclusion with Bounded Waiting by $4\log_2 N$-Valued Shared Variable". In: *Journal of Computer and System Sciences* 25.1 (1982), pp. 66–75.

[198] Sayra Ranjha et al. "Partial-order reduction in reachability-based response-time analyses of limited-preemptive DAG tasks". In: *Real-Time Systems* 59 (June 2023), pp. 1–55. DOI: 10.1007/s11241-023-09398-x.

[199] Anand S. Rao. "AgentSpeak(L): BDI agents speak out in a logical computable language". In: *Agents Breaking Away*. Ed. by Walter Van de Velde and John W. Perram. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 42–55. ISBN: 978-3-540-49621-2.

[200] A. J. Dinusha Rathnayaka and Vidyasagar Potdar. "Wireless Sensor Network transport protocol: A critical review". In: *J. Netw. Comput. Appl.* 36 (2013), pp. 134–146.

[201] Priyanka Rawat et al. "Wireless Sensor Networks: A Survey on Recent Developments and Potential Synergies". In: *J. Supercomput.* 68.1 (Apr. 2014), pp. 1–48. ISSN: 0920-8542. DOI: 10.1007/s11227-013-1021-9. URL: https://doi.org/10.1007/s11227-013-1021-9.

[202] Usman Raza, Parag Kulkarni, and Mahesh Sooriyabandara. "Low Power Wide Area Networks: An Overview". In: *IEEE Communications Surveys & Tutorials* 19.2 (2017), pp. 855–873. DOI: 10.1109/COMST.2017.2652320.

[203] Andrew Rebeiro-Hargrave et al. "MegaSense: Cyber-Physical System for Real-time Urban Air Quality Monitoring". In: *2020 15th IEEE Conference on Industrial Electronics and Applications (ICIEA)*. 2020, pp. 1–6. DOI: 10.1109/ICIEA48937.2020.9248143.

[204] M. Roggenbach et al. *Formal Methods for Software Engineering: Languages, Methods, Application Domains*. Texts in Theoretical Computer Science. An EATCS Series. Springer International Publishing, 2022. ISBN: 9783030387990. URL: https://books.google.co.uk/books?id=bH45zAEACAAJ.

[205] Fernando Royo, Miguel Lopez-Guerrero, et al. "2C-WSN: A Configuration Protocol Based on TDMA Communications over WSN". In: *GLOBECOM 2009 - 2009 IEEE Global Telecommunications Conference*. 2009, pp. 1–6. DOI: 10.1109/GLOCOM.2009.5425742.

[206] Fernando Royo, Miguel López-Guerrero, et al. "2C-WSN: A configuration protocol based on TDMA communications over WSN". In: Nov. 2009, pp. 1–6. DOI: 10.1109/GLOCOM.2009.5425742.

[207] M. Carmen Ruiz, Hermenegilda Macià, and Javier CALLEJA. "New Proposals to Improve a MAC Layer Protocol in Wireless Sensor Networks". In: *Informatica* 30 (Mar. 2019), pp. 91–116. DOI: 10.15388/Informatica.2019.199.

[208] Ahmed Yousuf Saber and Ganesh Kumar Venayagamoorthy. "Efficient Utilization of Renewable Energy Sources by Gridable Vehicles in Cyber-Physical Energy Systems". In: *IEEE Systems Journal* 4.3 (2010), pp. 285–294. DOI: 10.1109/JSYST.2010.2059212.

[209] Bahare Salmani and Joost-Pieter Katoen. "Fine-Tuning the Odds in Bayesian Networks". In: *Symbolic and Quantitative Approaches to Reasoning with Uncertainty*. Ed. by Jiřina Vejnarová and Nic Wilson. Cham: Springer International Publishing, 2021, pp. 268–283. ISBN: 978-3-030-86772-0.

[210] Mustapha Reda Senouci and Abdelhamid Mellouk. "1 - Wireless Sensor Networks". In: *Deploying Wireless Sensor Networks*. Ed. by Mustapha Reda Senouci and Abdelhamid Mellouk. Elsevier, 2016, pp. 1–19. ISBN: 978-1-78548-099-7. DOI: https://doi.org/10.1016/B978-1-78548-099-7.50001-5. URL: https://www.sciencedirect.com/science/article/pii/B9781785480997500015.

[211] Laya Shamgah et al. "Reactive Symbolic Planning and Control in Dynamic Adversarial Environments". In: *IEEE Transactions on Automatic Control* 68 (2023), pp. 3409–3424. URL: https://api.semanticscholar.org/CorpusID:251419122.

[212] Natarajan Shankar. "Verification by Abstraction". In: *Formal Methods at the Crossroads. From Panacea to Foundational Support: 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002. Revised Papers*. Ed. by Bernhard K. Aichernig and Tom Maibaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003,

pp. 367–380. ISBN: 978-3-540-40007-3. DOI: 10.1007/978-3-540-40007-3_23. URL: https://doi.org/10.1007/978-3-540-40007-3_23.

[213] Oliver Sharma et al. "Towards Verifying Correctness of Wireless Sensor Network Applications Using Insense and Spin". In: *Model Checking Software*. Ed. by Corina S. Păsăreanu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 223–240. ISBN: 978-3-642-02652-2.

[214] Jang-Ping Sheu, Jehn-Ruey Jiang, and Ching Tu. "Anonymous Path Routing in Wireless Sensor Networks". In: *2008 IEEE International Conference on Communications* (2008), pp. 2728–2734.

[215] Gagandeep Singh and Caterina Urban, eds. *Static Analysis - 29th International Symposium, SAS 2022, Auckland, New Zealand, December 5-7, 2022, Proceedings*. Vol. 13790. Lecture Notes in Computer Science. Springer, 2022. ISBN: 978-3-031-22307-5. DOI: 10.1007/978-3-031-22308-2. URL: https://doi.org/10.1007/978-3-031-22308-2.

[216] Stephen Smaldone et al. "The Cyber-Physical Bike: A Step towards Safer Green Transportation". In: *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. HotMobile '11. Phoenix, Arizona: Association for Computing Machinery, 2011, pp. 56–61. ISBN: 9781450306492. DOI: 10.1145/2184489.2184502. URL: https://doi.org/10.1145/2184489.2184502.

[217] Ana Sokolova and Erik P. de Vink. "Probabilistic Automata: System Types, Parallel Composition and Comparison". In: *Validation of Stochastic Systems*. 2004.

[218] Jianping Song et al. "WirelessHART: Applying Wireless Technology in Real-Time Industrial Process Control". In: *2008 IEEE Real-Time and Embedded Technology and Applications Symposium* (2008), pp. 377–386.

[219] "Static Analysis". In: *Reanalysis of Structures: A Unified Approach for Linear, Nonlinear, Static and Dynamic Systems*. Dordrecht: Springer Netherlands, 2008, pp. 1–36. ISBN: 978-1-4020-8198-9. DOI: 10.1007/978-1-4020-8198-9_1. URL: https://doi.org/10.1007/978-1-4020-8198-9_1.

[220] W. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton, 1994.

[221] Lei Tang et al. "PW-MAC: An energy-efficient predictive-wakeup MAC protocol for wireless sensor networks". In: May 2011, pp. 1305–1313. DOI: 10.1109/INFCOM.2011.5934913.

[222] Frank Tip. "A survey of program slicing techniques". In: *J. Program. Lang.* 3 (1994).

[223] Flavio Tonelli et al. “Cyber-physical systems (CPS) in supply chain management: from foundations to practical implementation”. In: *Procedia CIRP* 99 (2021). 14th CIRP Conference on Intelligent Computation in Manufacturing Engineering, 15-17 July 2020, pp. 598–603. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2021.03.080. URL: https://www.sciencedirect.com/science/article/pii/S221282712100370X.

[224] Güliz Tuncay et al. “Resolving the Predicament of Android Custom Permissions”. In: Jan. 2018. DOI: 10.14722/ndss.2018.23221.

[225] Ivaylo Valkov, Alastair F. Donaldson, and Alice Miller. “Synchronisation in Language-level Symmetry Reduction for Probabilistic Model Checking”. In: *30th International SPIN symposium on Model Checking of Software (SPIN 2024), Luxembourg* (Apr. 2024).

[226] Antti Valmari. “Stubborn sets for reduced state space generation”. In: *Advances in Petri Nets 1990*. Ed. by Grzegorz Rozenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 491–515. ISBN: 978-3-540-46369-6.

[227] Chaitanya Varma. “An Enhanced Algorithm for Variable Reordering in Binary Decision Diagrams”. In: *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 2018, pp. 1–4. DOI: 10.1109/ICCCNT.2018.8493495.

[228] Xavier Vilajosana et al. “6TiSCH: Industrial Performance for IPv6 Internet-of-Things Networks”. In: *Proceedings of the IEEE* 107 (June 2019), pp. 1153–1165. DOI: 10.1109/JPROC.2019.2906404.

[229] Yakir Vizel, Georg Weissenbacher, and Sharad Malik. “Boolean Satisfiability Solvers and Their Applications in Model Checking”. In: *Proceedings of the IEEE* 103.11 (2015), pp. 2021–2035. DOI: 10.1109/JPROC.2015.2455034.

[230] Mehmet Can Vuran and Ian F. Akyildiz. “XLP: A Cross-Layer Protocol for Efficient Communication in Wireless Sensor Networks”. In: *IEEE Transactions on Mobile Computing* 9 (2010), pp. 1578–1591.

[231] Thomas Wahl, Nicolas Blanc, and E. Allen Emerson. “SVISS: Symbolic Verification of Symmetric Systems”. In: *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. 2008.

[232] Thomas Wahl and A. F. Donaldson. “Replication and Abstraction: Symmetry in Automated Formal Verification”. In: *Symmetry* 2.2 (2010), pp. 799–847. DOI: 10.3390/SYM2020799.

[233] Changjing Wang and Jinyun Xue. “Formal Derivation of a High-Trustworthy Generic Algorithmic Program for Solving a Class of Path Problems”. In: *Frontiers in Algorithmics*. Ed. by Xiaotie Deng, John E. Hopcroft, and Jinyun Xue. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 27–39. ISBN: 978-3-642-02270-8.

[234] Chong Wang, Hai-ming Li, and Jia-jia Ye. “Software Simulation of Lifts”. In: *Network Computing and Information Security*. Ed. by Jingsheng Lei et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 315–322. ISBN: 978-3-642-35211-9.

[235] Zijian Wang, Eyuphan Bulut, and Boleslaw Karol Szymanski. “Energy Efficient Collision Aware Multipath Routing for Wireless Sensor Networks”. In: *2009 IEEE International Conference on Communications* (2009), pp. 1–5.

[236] M. Webster et al. “Formal verification of synchronisation, gossip and environmental effects for wireless sensor networks”. English. In: *EASST Electronic Communications* (2019). ISSN: 1863-2122. DOI: 10.14279/tuj.eceasst.76.1078.1045.

[237] Matt Webster et al. “Formal verification of synchronisation, gossip and environmental effects for wireless sensor networks”. In: *Electronic Communications of the EASST* 76 (2019).

[238] Stephan Weyer et al. “Future Modeling and Simulation of CPS-based Factories: an Example from the Automotive Industry”. In: *IFAC-PapersOnLine* 49.31 (2016). 12th IFAC Workshop on Intelligent Manufacturing Systems IMS 2016, pp. 97–102. ISSN: 2405-8963. DOI: https://doi.org/10.1016/j.ifacol.2016.12.168. URL: https://www.sciencedirect.com/science/article/pii/S2405896316328397.

[239] Xin Xin et al. “Dynamic probabilistic model checking for sensor validation in Industry 4.0 applications”. In: *2020 IEEE International Conference on Smart Internet of Things (SmartIoT)*. IEEE. 2020, pp. 43–50.

[240] Li Da Xu, Wu He, and Shancang Li. “Internet of Things in Industries: A Survey”. In: *IEEE Transactions on Industrial Informatics* 10.4 (2014), pp. 2233–2243. DOI: 10.1109/TII.2014.2300753.

[241] Shuang-Hua Yang. *Wireless Sensor Networks: Principles, Design and Applications*. 1st. Springer Publishing Company, Incorporated, 2016. ISBN: 1447169328.

[242] Yang Yang et al. “Software Simulation”. In: *5G Wireless Systems: Simulation and Evaluation Techniques*. Cham: Springer International Publishing, 2018, pp. 157–233. ISBN: 978-3-319-61869-2. DOI: 10.1007/978-3-319-61869-2_4. URL: https://doi.org/10.1007/978-3-319-61869-2_4.

[243] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. “Wireless sensor network survey”. In: *Computer Networks* 52.12 (2008), pp. 2292–2330. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2008.04.002. URL: https://www.sciencedirect.com/science/article/pii/S1389128608001254.

[244] Håkan L. S. Younes. “Ymer: A Statistical Model Checker”. In: *International Conference on Computer Aided Verification*. 2005.

[245] Håkan L. S. Younes and Reid G. Simmons. “Probabilistic Verification of Discrete Event Systems Using Acceptance Sampling”. In: *Computer Aided Verification*. Ed. by Ed Brinksma and Kim Guldstrand Larsen. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 223–235. ISBN: 978-3-540-45657-5.

[246] Theodore Zahariadis et al. “Efficient Detection of Routing Attacks in Wireless Sensor Networks”. In: July 2009, pp. 1–4. DOI: 10.1109/IWSSIP.2009.5367775.

[247] Pamela Zave. “Lightweight Verification of Network Protocols : The Case of Chord”. In: 2009.

[248] Yun Zhou, Yuguang Fang, and Yanchao Zhang. “Securing wireless sensor networks: a survey”. In: *IEEE Communications Surveys & Tutorials* 10.3 (2008), pp. 6–28. DOI: 10.1109/COMST.2008.4625802.