CHALMERS
UNIVERSITY OF TECHNOLOGY

Data Minimization in Distributed Applications for More Privacy

Master's thesis in Algorithms, Languages and Logic

JAKOB BOMAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016:NN

Data Minimization in Distributed Applications for More Privacy

JAKOB BOMAN

Department of Computer Science and Engineering
Division of Software Technology
Chalmers University of Technology
Gothenburg, Sweden 2016

Data Minimization in Distributed Applications for More Privacy
JAKOB BOMAN

Supervisor: Thibaud Antignac, Software Engineering
Examiner: Wolfgang Ahrendt, Software Engineering

Master's Thesis 2016:NN
Department of Computer Science and Engineering
Division of Software Technology
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2016

Data Minimization in Distributed Applications for More Privacy
JAKOB BOMAN
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

With the world becoming more and more digitalized, the collection of personal data from users becomes a growing concern. When data is collected and aggrevated to be accessed remotely, it makes it easier for adversaries to access information in a low-risk and anonymous manner. Data Minimization is a practical way to prevent leakages of personal data, to not collect more than is directly relevant, but it's a practice rarely used.

In this thesis we will investigate Over-Collection in a commonly used technology namely Wireless Sensor Network , and use Data Minimization as a tool to prevent this. We will describe our process of using Model Checking, present some revisions it yielded and the resulting Promela models. Even though this project sought to achieve a more extensive model, it's findings can still be useful for related work into Model Checking and Data Minimization.

Keywords: Formal Verification, Data Minization, Model Checking, Decision Procedures, Promela, Spin

Acknowledgements

First of all I want to thank David Frisk for this outstanding LaTeX-template I used for my master thesis.
Secondly I want to thank Thibaud Antignac for the idea of this thesis and his invaluable help during the majority of this thesis. Furthermore for behind a key role as a discussion partner and increasing my knowledge in this area, I hope you're having a good time back in France! Also I want to thank David Sands for stepping up and being my supervisor the last months of my project.
Lastly I want to thank Wolfgang Ahrendt, in his role as an examiner for this project and for making it possible.

Jakob Boman, Gothenburg, April 11, 2017

Contents

List of Figures

List of Tables

Listings

1 Introduction

This chapter will give a brief introduction to the thesis, introduce the motivation for why this thesis is relevant and then also present the aims sought to be achieved at the end.

1.1 Motivation

The presence of connected devices in our environment is increasing. These devices form a network often called Internet of Things (or IoT for short), where everything from lightbulbs to thermostats can be controlled by an app or by another device. These services make a lot of that data available to the end user but also to malicious parties due to the devices leaking more data than intended or by bad design. This puts the end user at risk, violating its privacy and leaking sensitive data.
One simple and obvious way to prevent leakages and misuses of personal data is to collect less of this data, a principle known as data minimisation. However, this solution is rarely used in practice because of business models relying on personal data harvest on one hand and because of the difficulty to enforce it once it is defined what is actually needed to provide a service.
Privacy is utterly important for the development of IoT applications, Miorandi et al. gives several reasons[1]: "The main reasons that makes privacy a fundamental IoT requirement lies in the envisioned IoT application domains and in the technologies used. Healthcare applications represent the most outstanding application field, whereby the lack of appropriate mechanisms for ensuring privacy of personal and/or sensitive information has harnessed the adoption of IoT technologies."

1.2 Aim

This thesis will investigate ways to improve privacy in a special kind of IoT (Internet of Things) devices known as Wireless Sensor Networks (WSN). WSN are networks of autonomous sensors and actuators. The goal to enhance privacy for this kind of devices will be addressed by relying on data minimization. This means the project sought to improve privacy in distributed networks by limiting the amount of personal data being processed.
To achieve this, the project sought to accomplish the following steps:

- Construct a model of a Wireless Sensor Network that illustrates Over-Collection.

- Investigate Over-Collection and by using Data Minimization, how it can be prevented.
- Implement a solution from the models and analyze results.

1.3 Limitations

The project will not consider faulty behaviors of a Wireless Sensor Network , meaning that the systems and algorithms will work under the assumption that all messages sent are received and all units are working as intended without malfunctions. Only the result of the data collection will be analyzed in the sought outcome and if time complexity of the algorithm will be an issue for the project, it will not be considered as a failure should it arise. Some analysis will be done but it won't be a main focus to minifor the project.
Only the privacy aspects of collecting personal data will be considered throughout the thesis and the aspect of storing and managing it will be outside the scope of this thesis.
Any model properties related to time (in the sense that they can be measured numerically, not in the sense of ordering events) will be treated on an abstract level or be disregarded. This practice is commonly used when modelling concurrent or distributed systems, from Ben-Ari (2008, p. 173); "Algorithms for these systems are designed to be independent of the speed of execution of a process or the speed at which a message is delivered, so it is sufficient to know that there are no errors caused by interleaving statements and messages."

1.4 Thesis Structure

Some abbreviations are used to explain the thesis structure, these will be explained later on as they are used in the thesis. The ordering of the remainder of the thesis is as follows:

Chapter 2 - Background presents the setting for the thesis and explains e.g. Data Minimization, Wireless Sensor Network . Also presents some related work to this thesis.

Chapter 3 - Method presents the method chosen for reaching the stated aim.

Chapter 4 - Theory presents an introduction to the tools chosen to reach the aim e.g. SPIN/Promela, Model Checking, Decision Procedures.

Chapter 5 - Modeling & Specification presents the modelling of the Wireless Sensor Networks and their specification.

Chapter 6 - Design presents results of the modelling, also argues their translation from their FSAs and shows some refinements the the Model Checking yielded.

Chapter 7 - Discussion presents some suggestions for future work.

Chapter 8 - Conclusion concludes the thesis.

2 Background

In this chapter we introduce the basic concepts that are the focus of this thesis, such as Wireless Sensor Networks and *Data Minimization*. It should provide an explanation to the technologies and issues that serve as the motivation for the thesis.

2.1 Wireless Sensor Network (WSN)

Wireless Sensor Networks are a growing interest in the research community [2, 3, 4]. The attention drawn to them can be motivated by new applications that are enabled from these large-scale networks of small devices that can collect information from a physical environment.
A Wireless Sensor Network is an improvement from the traditional sensor networks, made possible by advances in micro-electro-mechanical systems (MEMS) technology[5, 4] making sensor nodes that are smaller, multifunction and cheaper in comparison to previous sensors.

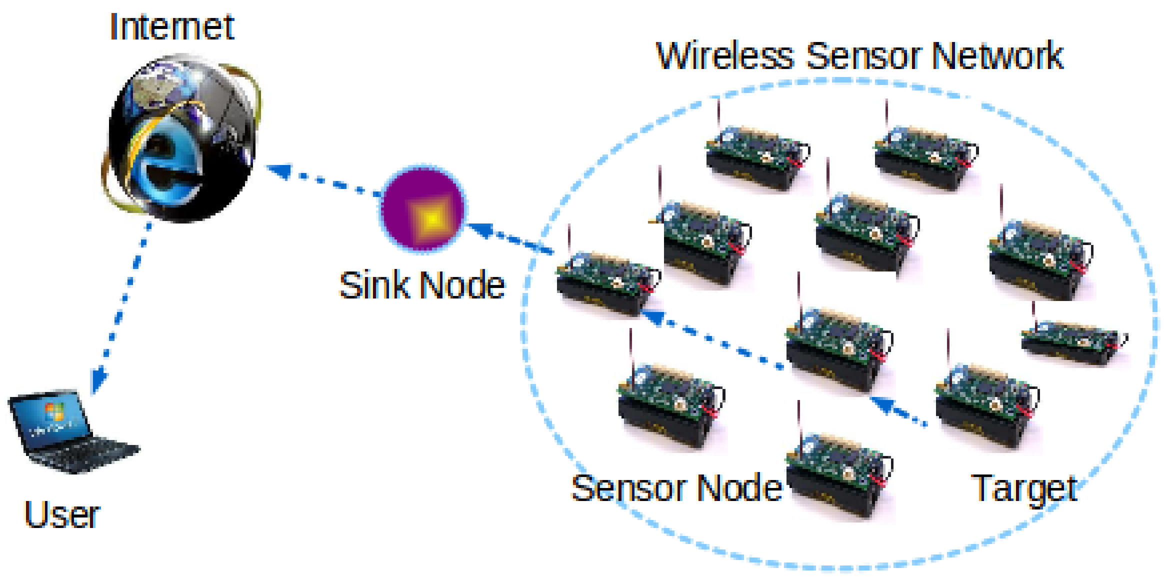


Figure 2.1: An illustration of a Wireless Sensor Network

Traditional sensors have two ways of being deployed; 1) They were positioned far away from the actual *phenomenon* (e.g. something known by sense perception) which required large sensors using complex techniques to distinguish the targets from surrounding noise. 2) Several sensors were deployed that only performed sensing and their communication topology had to be carefully engineered and they transmitted time series of the data to the central nodes which performed the communication.Wireless Sensor Networks on the other hand, is constructed by deploying a large number of sensor nodes close to the phenomenon and their position doesn't need to be engineered or predetermined[2].

Sensor networks can be built up by many different types of sensors such as acoustic, seismic, infrared, thermal, radar or visual, which can monitor an assortment of environmental conditions[6], e.g. temperature, humidity, pressure, noise levels.

These features that Wireless Sensor Networks supports, makes them available for a wide range of usages. For example, the city of Chicago began a project[7] to install Wireless Sensor Network for tracking information on urban conditions. The sensors are planned to be low-resolution cameras, infrared cameras and microphones for analyzing urban areas and sending the information back to a secure remote server. The article stresses that even though the sensors would have a low-resolution cameras, the video could still be used to identify individuals. Which could cause a privacy concern. To prevent this, that project will also define a privacy policy before installing the sensors, through public hearings and also informing the public what kind of information and how the information would be gathered.

There are many other practical applications for Wireless Sensor Networks some examples include environmental applications; for tracking small animals and birds, monitoring air pollution and precision agriculture[8, 9, 10]. In military applications for mission and flight systems in situ sensing and self-adapting mine-fields[11, 12]. Also in health-care applications for drug administration and large-scale monitoring of stress in the field[13, 14].

2.2 Privacy Issues in WSNs

When adversaries can access to sensitive information in Wireless Sensor Network , it's a privacy issue. This can either be achieved by accessing sensor data or eavesdropping on communications in the network. For example, if an adversary gains access to data from sensors monitoring a home on both the inside and the outside, they could derive information regarding the inhabitants' behaviours or private activities.

Furthermore, there are many other issues related to privacy. In a paper by Haowen and Perrig[15], they state that the main problem related to privacy is that the information from sensor nodes are aggrevated and can be accessed remotely. This makes it easier for adversaries to access information in a low-risk and anonymous manner. Also making it possible for just one adversary to monitor multiple sites at once.

Finally, a major issue was mentioned in an paper by Al Ameen, Liu and Kwak[16], as privacy is a major concern in healthcare applications. They claimed that if the privacy issues aren't honestly debated, there is a risk for public backlash that will result in mistrust and might make available technology not used. This can happen either if the data is obtained with or without the consent of the person, since the damage can happen either way.

2.3 Data Minimization

As defined by the EDPS (European Data Protection Supervisor); "The principle of data minimization means that a data controller should limit the collection of personal information to what is directly relevant and necessary to accomplish a specified purpose."[17]
With the world becoming more and more digitalized, the collection of personal data from users becomes a growing concern. Today, data processing entities automatically makes decisions based on data analysis which can impact the lives of individuals, which in turn makes the need for protecting their personal data is even greater.[18] Another problem which arises from being more connected is that individuals, whose data is being collected, are often unaware of the consequences of the data processing that comes after. Also, legal repercussions for infringement of data protection obligations is usually only due to a breach or a misuse that has already occured. There are systems that support this thinking, that requires users to know what data is being collected, such as Privacy Enchaning Information Management Systems (PE-IMS) which will be discussed later on in this chapter.

In a paper by Pfitzmann, Andreas and Hansen and Marit, a combined terminology for the aspects of Data Minimization was defined. The main definitions they used were: *Anonymity*, *Unlinkability*, *Undetectability* and *Unobservability*[19]. These definitions sought to explain data minimization. To explain these definitions, we will use the same terminology as in the paper, where the two most important ones are *subjects* and *Items of Interest* (IOI).
For privacy reasons, being that we wish to maximize it for human beings, subjects are mainly users in a system. But in the definitions that follow, it can be a legal person or even be a computer. An Item of Interest is a generalization of what can be seen as information, e.g. the contents of a message, the name or the pseudonym of a user or even the action of a user sending a message. All of these can be an Item of Interest, and the definitions are stated in the following list:

Anonymity means that for a subject to have anonymity, it has to be indistinguisable in a set of subjects. Meaning that if a subject has a set of attributes defining the subject, there always has to exist an appropriate set of subjects with potentially the same attributes, in other words: "Anonymity of a subject means that the subject is not identifiable within a set of subjects, the *anonymity set.*", where the anonymity set is the set of all possible subjects. The opposite of anonymity is called *Identifiability.*

Unlinkability is the relation between IOIs in the system. If several IOIs become compromised, an attacker should not be able to distinguish whether these IOIs are related or not. The opposite of unlinkability is called *Linkability.*

Undetectability is an attribute for an IOI. It means that an attacker should not be able to distinguish whether the item exists or not. The opposite of undetectability is called *Detectability.*

Unobservability is an attribute which is combination of *anonymity* and *undetectability.* It means that, in regards to an IOI, that neither if a subject is involved in the IOI or not, he or she should be aware of the other involved subjects. The opposite of unobservability is called *Observability.*

2.3.1 Over-Collection

In contrast to Data Minimization, there is a term called Data Over-Collection. In a paper by Yibin Li et al**??**, they adressed issues of data over-collection and presented a privacy protection framework to solve them. Their focus was the "smart city", an urban development vision with social facilities being connected wirelessly with information and communication technology and IoT-technology in a secure fashion to improve the efficiency of services. The system would help identify users in a smart fashion to relieve the need of ID- or credit cards. This requires that smart cities have a system that contains a lot of different users' information, which puts the users at a potential privacy leakage when accessing these features. They also sought to prevent over-collection of data, but though a privacy protection. Their definition of data over-collection was; "Collecting data more than enough on it's original function while within the permission scope". This definition is related to that their target, for which they sought to minimize over-collection, was smartphones. An example they used for explaining this definition was the following: "I take a picture and want to share it with my friends via some SNS app in my smartphone. For sharing this photo to my friends, I have to agree the permission request from this app. After authorize the access permission to this SNS app, all my photos are available to this app, but I only want this app to access one specific photo. As a result, this app may collect data more than enough on my original requirement while within the permission scope which I authorize."

2.4 Modelling Concurrency

In this section we will present some different approaches to modelling concurrency. These will give an insight into the other tools available, the chosen approach for this thesis (Model Checking) will not be described here but will be thoroughly explained in the next chapter.

2.4.1 Petri Nets

In 1962, Carl Adam Petri disserted his work on a more graphical way of modelling concurrency, namely using something called *Petri Nets.* A Petri Net is a directed bipartite graph. An example can be seen in Figure 2.2. Each node in the graph represent transitions, shown as bars, or places, which are shown as circles. The black dots at places are called tokens, they indicate the holding of a condition at that place.

Connecting the nodes are directed arcs that describe pre- and/or postconditions for the transitions, illustrated as arrows.

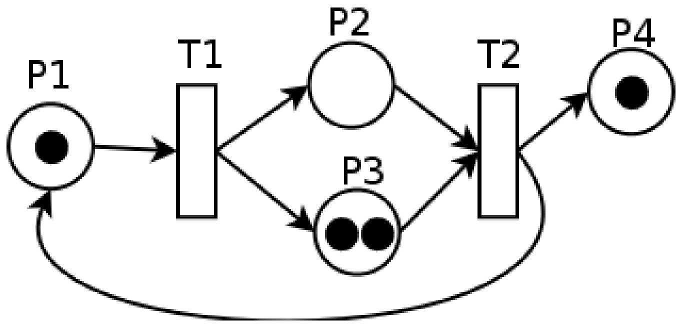


Figure 2.2: Illustration of a Petri Net

The primary rule for Petri Net theory is the rule for transition enabling and firing. It derives from the idea that many systems can be described as system states and their changes. So to simulate the behavior of a system, each state or marking are allowed to change according to the following rule:

- A transition is enabled iff each of its input places has atleast one token
- A transition can only fire if it is enabled.
- When a transition fires it removes a token from each of its input places and a token is deposited into each of its output places.

By analyzing different properties of a Petri net model of a system, such as liveness and boundedness, one can prove other properties aswell. For example, a Petri net is said to be *live*, if there always exist a fire sequence to each transition in the model, and if the model is live it's also guaranteed to be free of deadlock[20]. A Petri net is said to be *k-bounded* if for each place, there exists an upper bound *k*, for how many tokens that can be there simultaneously. If *k* is 1, then the system is said to be *safe*.

2.4.2 Process Calculi

Process Calculi is a family of related approaches for formal modelling of concurrent systems. It allows for a high-level description of the interaction, communication and the synchronization between processes and agents. Some examples of different Process Calculi are CSP, LOTOS and π-calculus[21]. Their focus vary, for they are specialized on modelling different systems, but some features they (and other Process Calculi) share are:

- Interaction between processes are represented as communication(message-passing), rather than manipulation of shared variables.
- Processes are described as a collection of primitives and operators for those primitives.

- Algebraic laws are defined for the process operators, which allows them to be analyzed by *equational reasoning*.

Initially, to define a *process calculus*, you start with a set of channels as a means of communication. The internal structure of channels are rich and are constructed to improve efficiency, but when explained theoretically these improvements are usually abstracted away. Also, a way to form new processes from old ones is required, this also varies from the different implementations but what they have in common can be summarized the following:

- A way of expressing parallell composition of processes
- A way of specifying which channels are used for sending and receiving data
- A way to sequentialize interactions
- A way to hide interaction points
- Recursion or a way to process replication

Furthermore, an example of constructing a process calculus model can be seen in the CRC Handbook of Computer Science and Engineering[22], written in π-calculus and compared to λ-calculus.

2.5 Related Work

This section will present some related work into also ensuring privacy for users, where other approaches than the one in this thesis were used.

2.5.1 Data Erasure and Declassification

Another step to consider when ensuring privacy for users having their data collected is the aspect of releasing or removing the data. In many systems, both of these functionalities are required. For this sake, different policies for *erasure* and *declassification* need to be clearly specified so the users' privacy is protected.
In an paper by Chong and Myers[23], they propose a security policy framework where policies for both can be specified so it suits the desired application. In said framework, one would specify an erasure policy on under which conditions information must be erased. One could also state what policy that would allow data to survive erasure, since information could be allowed to still exist within a system in a restricted form. Secondly, declassification policies would define what policy should be enforced on new information, the conditions under which said information would be declassified and finally the policy it should have after declassification.
This approach covers an important aspect of privacy, namely the managing of personal data. This differs from the focus of the thesis, as we seek to minimize the collection of data to achieve better privacy.

2.5.2 Privacy Enhancing Identity Management Systems (PE-IMS)

In an online setting, it's assumed that people would like to retain their anononymity.[24] To let users manually control their identities would be a cumbersome process, so instead an automated solution managing this would be preferred. Such a solution can be an Identity Management System (IMS).
An IMS is a system that allows support for "administration of information subjects". An extension of this is Privacy-Enhancing Identity Management Systems (PE-IMS) which supports "active management of personal information" which grants all parties involved flexibility and control over their personal data. A principle used for this is called 'Notice and Choice', a central aspect of data minimization, which means user-controlled linkability of personal data. This puts the responsibility on the user to make informed choices of representing and managing their partial identities.
This allows a user to be as anonymous as they wish, within the predefined limits, since a PE-IMS can be designed to offer any degree of anonymity and linkability. Applications utilizing PE-IMS would specify the range of choices available to the user. Some applications might require some authenticity from the user, e.g. government processes, and in other cases a user could be allowed complete anonymity. By allowing each application different levels of authorization, one can minimize linkability between different communication events and still maximize information exchange while preventing context-spanning profiling.

3
Method

In order to reach the stated aim, as posed in Section 1.2, this project is divided into two phases:

Phase 1: An initial study of the tools available to model a Wireless Sensor Network , where one will be chosen to be used. We will then seek to formalize the characteristics of a Wireless Sensor Network , to help the future analysis of the system.
From this we will proceed with an agile development process, and construct a system model of a Wireless Sensor Network that can over-collect data. This system model will then be abstracted and refined to provide an analysis of a general system collecting data. This means we will abstract away any characterstics of a system that we consider irrelevant for our analysis. Furthermore, we will proceed with the analysis:

1. How and when is Over-Collection occuring in a system?
2. How can we apply Data Minimization to our model to prevent this?

With this analysis completed, we will proceed with an implementation of the models in a suitable programming language to achieve a prototype.

Phase 2: We will then extend our models further, to model a more extensive example of a Wireless Sensor Network collecting data. Here we are open to modelling specifics of a system and abstractions will only be done if required.

With this we will proceed with the same analysis as mentioned in Phase 1, the findings from the previous analysis will serve as a framework for this analysis.

4 Theory

This chapter provides an introduction to different tools that were used in the thesis to reach the stated aim. First we explain *Formal Verification*, following it up with *Model Checking* and its' strengths and weaknesses. Also some introduction to the model checking tool *SPIN*, Simple Promela INterpreter, and its' input language *Promela*.

4.1 Formal Verification

The act of formal verification means to make use of mathematical techniques to make sure that a design upholds a defined functional correctness [25]. This means, that if we assume we have the following: a model of a design, a description of the environment where the design is supposed to operate in and some properties we wish the design to uphold. With this information, one may want to construct some input sequences, that are in the allowed in domain of the environment, that would violate the properties stated. A common practice for finding such patterns today are random simulations or directed tests. Formal verification allows for an extended approach to this, as it allows both to search for input sequences that violates the properties but also allows to mathematically prove that the stated properties holds when no input sequences exist.

4.2 Model Checking

A traditional approach to verifying concurrent systems is based on using extensive testing and simulation to find and eliminate unwanted occurrences from the system, but this way can easily miss crucial errors when the system that's being tested has a large number of possible states[26]. An alternative technique that was developed in the 1980's by Clarke et al. is called *temporal logic model checking* or "Model Checking".
Model Checking is an automated technique to verify finite state concurrent systems, by letting a tool verify that a model holds for certain properties. The process of applying Model Checking to a design is separated into several tasks; *modeling*, *specification* and *verification*.

> **Modeling:** First task is to translate a design into a format which is accepted by a model checking tool. This is either a compilation task or a task in abstracting certain aspects of the design to eliminate irrelevant or unimportant

details, due to limitations on time an memory.

Specification: Second task is to state which properties the design is supposed to have. This is usually done using in a logical formalism, commonly in temporal logic, which can express assertions on a system evolving over time.

Verification: The final step is allowing the tool to verify the specification on the model. This will either be a positive result, meaning the model satisfies the properties, or a negative result where the properties aren't. A negative result can also be that the model's state space is too large to fit into a computer, which will require the model to be further abstracted to be verified.

4.2.1 Model Checking Workflow

The use of model checking in practice typically follows the workflow in Figure 4.1. A design is translated into a description, that the model checker can read, and a specification of wanted or unwanted behavior is translated into a property. Then the model checker will produce a result which is either that the property is upheld or an error explaining how the property is invalidated[26].

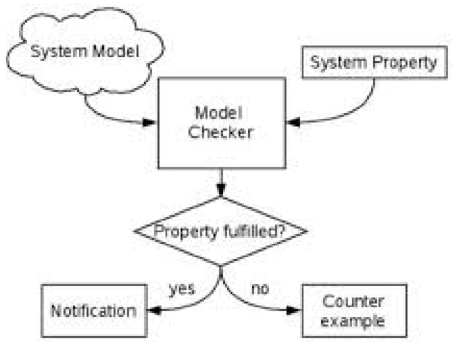


Figure 4.1: Workflow of Model Checking

4.3 SPIN

The model checking tool used in this thesis is called SPIN, an abbreviation of Simple Promela Interpreter. The SPIN tool allows to create an abstract model of a system, specifying properties that the model must hold and then verify them to see if there is possible system state that invalidates it. SPIN verification models are focused on proving the correctness of process interactions.[27] Process interactions can be specified in several ways using SPIN; rendevouz primitives, asynchronous message passing, shared variables or a combination of these.

4.3.1 Promela

Promela is a specification language with its' focus on modeling process synchronization and coordination rather than computation. Therefore the language targets the description of conurrent software systems, rather than the description of hardware circuits, which is more common for other model checking applications[27]. The features in the Promela language allows for description of concurrent processes and communication through message passing over buffered or rendevouz(unbuffered) channels.

Promela Example

To give an impression of Promela's syntax, Listing 4.1 serves as an small example that captures most of the concepts used in this thesis. The example models an procedure called *environment*, receiving a message `meter` on the channel `envChan`. Then the process undeterministically choses one of the two responses in the guard statement and responds back on the same channel. Worth noting is that most part of the model is captured in an `atomic`-statement, this means that when the request is received, this process will be allowed to execute the rest of the `atomic`-statement without any interleaving. Since this process flow isn't realistic in a concurrent system, where interleaving is prone to occur, all usage of `atomic` has to be explained and carefully motivated.

Listing 4.1: Promela Example

```
active proctype Environment() {

Idle:
  if
  :: atomic {
    envChan ? meter ->
      if
      :: envChan ! bigData;
      :: envChan ! smallData;
      fi;
      goto Idle;
  }
  fi;
}
```

4.3.2 Properties in SPIN

Specification

In order to prove or disprove a property using SPIN, we must first state them in some formal notation**??**. This can be done by either using `assertion`-statements, to ensure a property at a certain point in time, or using LTL to prove properties over an entire system trace. Except the operators inherited from propositional logic (*negation*, *conjunction*, *equivalence*, *implication*, etc.) LTL also provides the temporal operators such as *always*, *eventually* and *until*.

Always ($\Box$) states that a property has to hold on the entire subsequent path, e.g $\Box a$ means that the condition a always holds true. In promela this is either written as `always` or `[]`.

Eventually ($\Diamond$) states that a property has to hold somewhere on the subsequent path, meaning that $\Diamond a$ means that a must hold in the current state or in some future state. This is written in promela as `<>` or `eventually`.

Until (U) captures a relative behavior between two condditions, e.g. a U b means that a must hold true atleast until b holds true. In promela this is written as `U` or `until`.

For a complete description of Linear Time Logic and its' semantics in SPIN, see Holzmann (2003, p. 135-139).

Verification

Spin allows us to either prove properties that always should hold true (safety properties) or error behaviors (i.e. properties that should never hold). When verifying safety properties in Spin, instead of trying to prove that a property holds true in each possible system state, it tries to find a state in which the property is invalidated. This is intuitively a faster way of finding erroneous behavior since when the verification finds one counterexample to the stated property, it no longer needs to search other states. So when running the verification, Spin negates the specified property and then attempts to find a system trace in which the negated property holds. If this is successful, then the property can be violated. Otherwise, if no such trace exists, the property is verified to always hold true.

4.3.3 Problem space reduction

There are two strategies that SPIN uses to reduce the number of states generated in verification **??**. The aims of them are; "to reduce the amount of reachable system states that must be searched to verify properties, or to reduce the amount of memory that is needed to store each state".

One strategy to achieve this is called partial order reduction, which relies on selecting and examining only a subset of all possible execution paths. An example of this is by detecting interleaving of processes which relative ordering do not affect the final outcome of the execution, with regards to the property being verified.

Another strategy SPIN uses is stutter equivalence. As explained by Peled, Wilke & Wolper in 1995[28]: “a pair of sequences are considered to be equivalent if they differ in at most the number of times a state may adjacently repeat”. Spin's partial order reduction strategy assumes this, and by so only guarantees verification for stutter invariant properties. This makes it impossible to verify properties containing the *next*-operator**??**, meaning an LTL formula which doesn't use the *next*-operator is by guarantee also stutter invariant.

4.3.4 Decision Procedures

In our research of over-collection, the project realized we needed some algorithmic way for a system to determine if it was over-collecting or not. To this end, we studied *decision problems*. A decision problem is a question expressed in some formal system that can be stated a "yes-no" question. And an algorithm used for solving decision problems is called a *decision procedure*, which terminates with a "yes" or "no" answer[29].
Decision procedures has a wide range of uses and is increasingly being used in hardware verification and theorem proving tools[30]. Some examples of such decision procedures are Yices, SVC, CVC Lite, UCLID [31, 32, 33, 34]. Besides theorem-proving and hardware verification, decision procedures are also increasingly being used in large-scale program analysis, bug finding and test generation tools (e.g. [35, 36]).

5
Modeling & Specification

In order to address the goals stated in the aim, the project first had to visualize and construct the models which would be investigated. To this end, the project first devised a terminology for the modeling to simplify the analysis the system.

5.1 Definitions

First we will describe how a typical Wireless Sensor Network , that we sought to model, would look. The project begun with an example as simple and general as possible, having an agile development process in mind, which would then be expanded to more complex models later on.

5.1.1 Basic WSN

As mentioned in the background, a basic Wireless Sensor Network consists of a set of collection nodes (referred to as "nodes"), a central server (referred to as "Server") and finally an Environment (the observed source). An illustration of this can be seen in Figure 5.1.

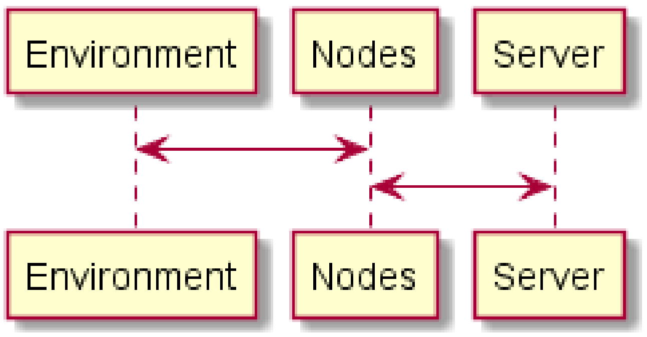


Figure 5.1: An illustration of a Wireless Sensor Network

These entities were chosen as the three pieces that our initial models would consist of. To abstract these further, the word *Actor* was used to represent that these entities were processes acting on data. Furthermore, the project followed by defining the characteristics of each of these.

5.2 Actors

A Wireless Sensor Network is built up by several different entities that communicates data between each other. Generally a network consists of multiples of virtually the same entity, e.g. multiple collection nodes, where each of these are running the different instances of the same process.
To describe the interaction between two actors in the system, behavior models were used (e.g. Figure 5.4). Where the name of each actor is shown in the boxes at the top. The message channel used between them is shown as the arrows, where the arrow-head points to the actor receiving the message and the contents of the message is referenced above it. Finally the ordering of the messages are in a ascending order from the top, meaning the first message sent is shown furthest to the top of the figure.
At this stage the project realized that to simplify the modeling process, the environment could be considered as an actor in the system. This was an abstraction, since the observed source wouldn't act from a predefined pattern as an actor would, but to save time from having to manage concurrency with a shared resource.

5.2.1 Server Actor

The server is an actor receiving messages from nodes and storing it for later usage. A server's behavior will vary depending on the structure of the system. If the decision is taken centrally the server will be the one checking for over-collection, otherwise it will be a node. Also if the communication is managed through the server, if the nodes doesn't communicate with each other, the server will act as a repeater for the decision.
In Figure 5.2 is the behavior for a system where server makes the decision and nodes doesn't communicate with each other. First, the node sends some data, the server checks for over-collection and replies accordingly. The response will either be a "stop" signaling that over-collection has occurred and the node should stop collecting or it tells it that it can continue collecting.
In addition to the behavior model, an Finite State Automaton(FSA) were designed for the server (Figure 5.3). The initial state being `Idle_a`, which the server will stay in until some `data` is received. "Data" being either `bigData` or `smallData`; $\texttt{data} \in \{\texttt{smallData}, \texttt{bigData}\}$. This abstraction will be further discussed in Section 6.1. The data is received in `Idle_a` and `Idle_s` and then checked in `Answ.` and `Hold`, meaning only the outgoing transitions from `Idle_a` and `Idle_s` is incoming data, the other are calculated internally. This also means the server will loop indefinetly between `Answ` and `Idle_a` as long as only smallData is received. When `bigData` is received the server will enter the `Idle_s-Hold` loop instead, which denotes the states where the server is requesting the nodes to stop collecting more data.

5.2.2 Environment Actor

The process for the environment actor had two steps:

1. Generate random data

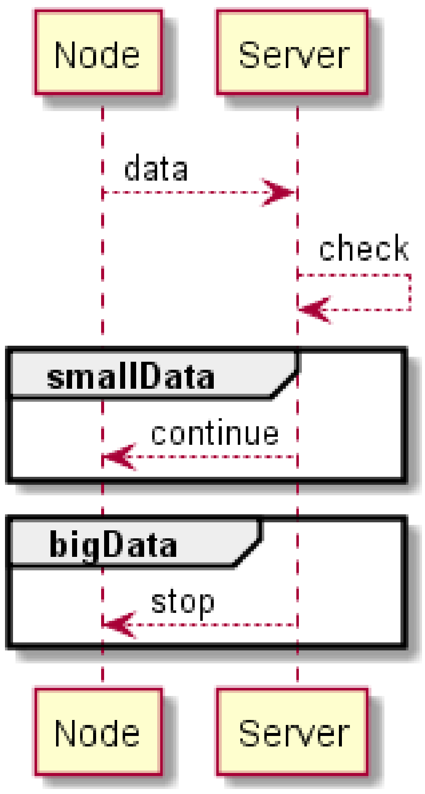


Figure 5.2: Behavior Model between Server and the Node

2. Serve random data to a requesting node

As mentioned before, the first step is not intuitive for an environment since the observed source isn't randomly varying, but for modeling purposes this is a simplification made to reduce the complexity of the model. In Figure 5.4 the behavior between a node and the the environment is described.

The corresponding FSA for the environment is seen in Figure 5.5. The environment will stay in the initial state `W` (short for "Waiting"), until a node `meter` it. Then the data is "generated" in `G` (short for "generate data") and served back to the node.

5.2.3 Node Actor

As seen in the behavior model for the node actor (Figure 5.6), it captures the majority of a typical scenario for the entire system. That is intuitive since the node communicates with both of the other actors of the system and is a intermediate part of the system. The scenario is when a node collects data, that doesn't cause a system-change, and forwards it to the server.

The alternative behavior for system is described in Figure 5.7 instead. There the data collected causes the server to make the decision that the node should stop collecting.

This behavior can be described in a FSA, as seen in Figure 5.8. The node meters data from the environment and passes it forward to the system. There it waits (noted by the state `Wait`) for a response before returning to the `Idle` state.

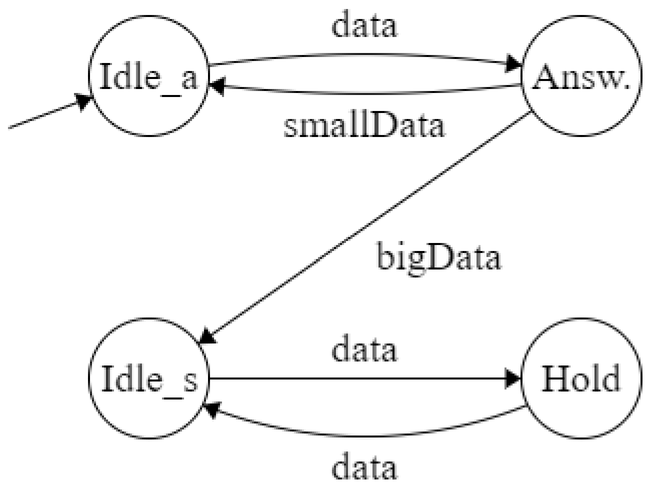


Figure 5.3: Finite State Automata for the Server Actor

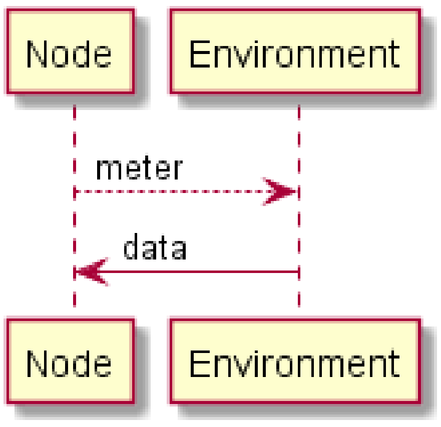


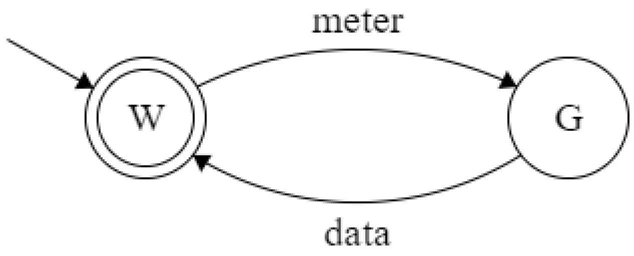
Figure 5.4: Behavior Model for the Environment

5.3 Modeling

With this defined, the project began to construct Promela models representing these actors' behaviour. This would be the initial example which then would be expanded step by step into a complex model. This section will describe how the modeling was carried out.

5.3.1 Initial Model

The first iteration consisted of a Wireless Sensor Network of three collection nodes, a server and an environment. The collection nodes would sent a request to the environment to receive data which then would be passed to the server for analysis. Each message being sent to the server could be the source of over-collection so to each message received, the server would respond to the nodes whether to continue collection or not. The promela representation of this can be seen in Appendix 1.2.



$$\begin{tikzcd} \arrow[r] & W \arrow[r, bend left, "meter"] & G \arrow[l, bend left, "data"] \end{tikzcd}$$

Figure 5.5: FSA for the Environment Actor

As can be seen in the example, the network passes the data between the actors and when the server triggers the decision to stop, the message is passed through the network and the system shuts down. To verify that this functionality was achieved, a correctness property was specified as *When the decision to stop is taken, the system should shut down.*

This initial model were then analyzed, for how it could be expanded to represent a more complex example. This analysis is described following subchapters.

5.3.2 Variations

During the work on the initial model, we realized that some assumptions were made for the system and that our initial model would only work if the target system communicated in the same way. Which might not always be true. And since we hadn't chosen a predefined system to model, we aimed to keep our model as general as possible to better investigate over-collection. So to abstract our model we composed a set of different architectural variations of Wireless Sensor Networks to model, which we considered to help us achieve the sought aim.
The variations the project chosed to focus on were the following:

Centralized or Decentralized decision making
The first choice reflected how much the sensor nodes would analyze the data. Since nodes can have a processing unit, they could potentially analyze the collected data and make a decision on their own.

Conjunctive or Disjunctive decision analysis
The second choice reflected how the decision were processed, if the data from a single data point could trigger a decision or if the decision considered data from multiple entries in it's evaluation.

Centralized or Distributed communication
The final variation reflected how the network communicated, it was considered centralized if all communication were sent through a central unit, such as a server, or if nodes were allowed to communicate independently to each other.

This meant that our already constructed model were a model with **centralized decision making**, **disjunctive decision analysis** and **centralized communi-**

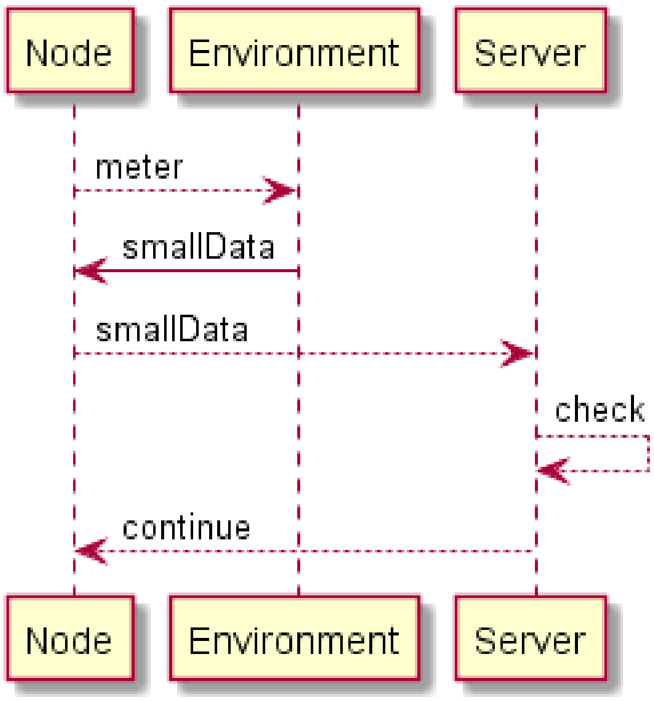


Figure 5.6: Behavior Model for a Node

cation. From this the project continued to refine a model for each of the other variations. During the refinement of the these models, several revisions were made before the end result was reached. The following section will explain some of these intermediate revisions that led to the final result.

5.4 Specification

This section presents the properties used to verify the system. The process of defining the properties were an iterative approach and several versions were considered, this section only covers the final properties were the result of the previously explained process.

5.4.1 Properties

The properties defined on the network were formulated using *Linear Time Logic* (LTL). This choice came from the fact that LTL were native to SPIN and the models were abstracted to only focus on the relevant parts to the project, LTL could provide a simple and direct specification to that property.

Correctness

The primary property sought of the system was, intuitively, that it was working as intended. This was formulated as a safety correctness property, to ensure that when

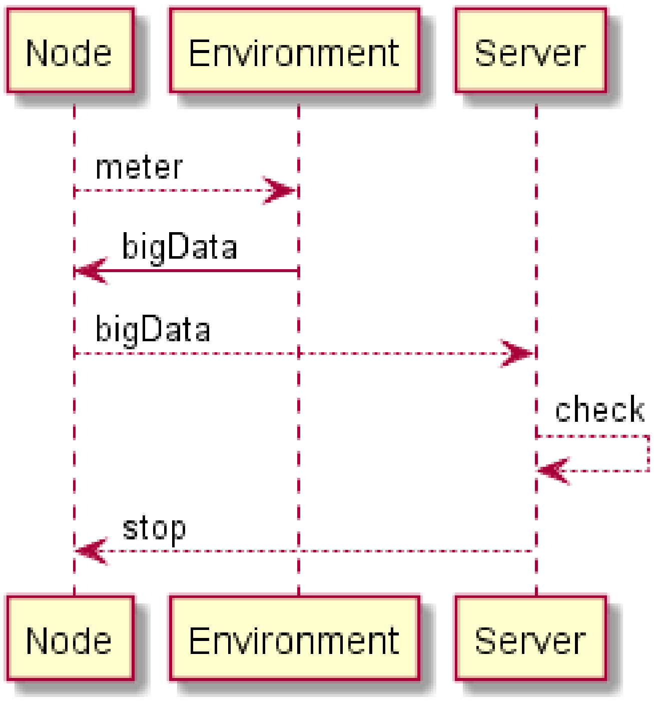


Figure 5.7: Behavior Model for a Node over-collecting

decision had been taken, the system respond to the by changing its' behavior.

Definition 1. Safety Correctness
When the stop-decision is taken, the system should stop collecting.

LTL: $\Box(O \rightarrow (\Diamond D))$

Where **O** and **D** corresponds to the state where the stop-decision is taken and the states where collection is stopped respectively. This captures the sought system change; whenever the system reaches the state O, eventually it will reach state D. An immediate change is not required, therefore the timing is relaxed by the eventually-operator.

Liveness

The second property that was specified was liveness, which expresses that "eventually something good happens". For our system, this was defined as "a node should only stops collecting if over-collection would occur".

Definition 2. Liveness
A node only stops collecting if over-collection would occur.

LTL: ¬ (Node_Done **U** Stop_Coll)

Where Node_Done denotes the state where the node no longer collects data and Stop_Coll denotes the state where the system takes the decision to stop collection.

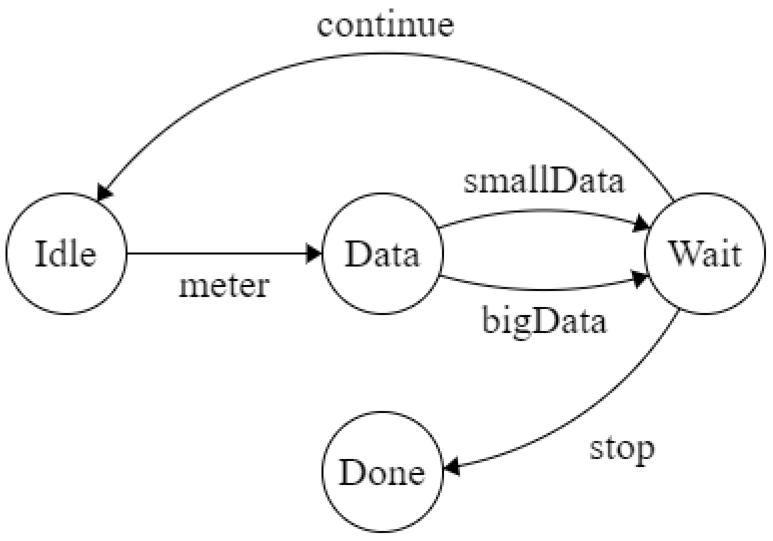


Figure 5.8: FSA for the Node Actor

U denotes the **Until**-operator.

6
Design

This section will present the resulting design of the Promela models, explain the systematic translation from the FSA models from Section 5.2 and also give some examples of the refinements that the Model Checking yielded.

6.1 System Description

The system consists of message channels between three different classes of procedures **Node**, **Server** and **Environment**. Each of these corresponds to the *Actors* specified in Section 5.2.
As an abstraction, the project considered the data sent in the network as a set of two possibilities. Either the data collected by the system causes the system to take the decision to stop collecting, to prevent over-collection, or it doesn't and it continues as before.

$$\text{data} \in (\text{smallData},\text{bigData})$$

This were noted as `bigData` and `smallData`, where `bigData` causes the system change and `smallData` doesn't. This simplication was made to reduce the state space, and the abstraction was sound as any data collected by the system would result in either of the two events.
The system procedures communicate using shared communication channels, `envChan` for the communication between the `Nodes` and `Environment` and `servChan` for the communication between the `Server` and the `Nodes`. This functionality is what is required for the in the behaviour models as shown in Section 5.2. Continuing on we will cover the actors indepdendently in the subsequent sections.

6.1.1 Environment

As mentioned previously, the environment was abstracted to also be an actor to simplify the work. If this wasn't the case, it would be considered a shared resource between the nodes where each node can individually meter the environment and then communicate it to the server. So to translate this, the environment is constructed as an atomic statement so when a node puts up a request on the channel it's instantly replied to before any other statement is executed. This removes the issue of interleaving which couldn't occur in reality. To handle randomness from the environment, so the entire set `data` was reachable, an `if`-statement without guards is used, which is through model checking guarantees the reachability.

The FSA of the environment consisted of two states, as seen in Figure 5.5, `Waiting` and `Generating Data`. Initially remained in the `Waiting` and when a it received a `meter`-input it transitioned to `G`, responded with `data`, and transitioned back. The language of this translates to the first part of the code, the procedure remains on listening on the channel `envChan` before receiving a `meter`. The atomic statement that captures it, doesn't halt the the execution as atomic statements in promela is built to only be active if the first statement in them are executable. Only once the `meter` is received, it becomes active. The following `if`-statement corresponds to the `Generating Data`-state, which responds with data∈(smallData,bigData). The final statement of the atomic statement is to return to the label `Idle`, which corresponds to the transition back to `Waiting`.

Listing 6.1: Environment code

```
active proctype Env() {

Idle:
        if
        :: atomic {
             envChan ? meter ->
               if // random outcome
               :: envChan ! bigData;
               :: envChan ! smallData;
               fi;
             goto Idle;
           }
        fi;
```

6.1.2 Server

The server consists of four states, where two of behaves the same. Initially the program remains in `Idle_Answering` (which corresponds to `Idle_a`). The model will listen to input from the different nodes in turn, when data is received it will transition to `Answering`. If smallData was received, the model would transition back to the previous state. If bigData ever was received by the server, the FSA should transition to `Idle_Stopping/Idle_S`. From here the FSA could only transition to `Hold`, which corresponds to the label `Stopping`.

Listing 6.2: Server code

```
active proctype Server() {

chan active_chan;
int i=0;

Idle_Answering:
          if
          :: nempty(servChan[i]) ->
               active_chan = servChan[i];
               goto Answering;
          :: empty(servChan[i]) ->
               i=(i+1)%NUM_NODES;
               goto Idle_Answering;
```

```
        fi;

Idle_Stopping:
        if
        :: nempty(servChan[i]) ->
            active_chan = servChan[i];
            goto Stopping;
        :: empty(servChan[i]) ->
            i=(i+1)%NUM_NODES;
            goto Idle_Stopping;
        fi;

Answering:
        if
        :: active_chan ? smallData ->
            active_chan ! continue;
            goto Idle_Answering;
        :: active_chan ? bigData ->
            active_chan ! stop;
            goto Idle_Stopping;
        fi;

Stopping:
        if
        :: active_chan ? smallData ->
            active_chan ! stop;
            goto Idle_Stopping;
        :: active_chan ? bigData ->
            active_chan ! stop;
            goto Idle_Stopping;
        fi;
}
```

6.1.3 Node

The node is initialized with the channel it communicates to the server with. It starts by metering the environment, then communicates it to the server and proceeds into `Waiting` to wait for an answer.
This corresponds the `Idle`-state of our FSA, as seen in Figure 5.8. The state `Data` is translated into the `if`-statement, where the same data is forwarded. The model then transitions to `Wait`, which translates to the jump to label `Waiting`. In the Waiting state the model either returns to `Idle` if it receives `continue` or transitions to `DoneColl` upon receiving `Stop`, which corresponds to `Done` in the FSA.

Listing 6.3: Node code

```
proctype Node(chan out) {
Idle:
        envChan ! meter;
        if
        :: envChan ? bigData ->
           out ! bigData;
           goto Waiting;
        :: envChan ? smallData ->
```

```
                out ! smallData;
                goto Waiting;
            fi;
Waiting:
            atomic {
              if
              :: out ? continue -> goto Idle;
              :: out ? stop ->
              fi;
            }
DoneColl: // node will no longer collect upon reaching this state
}
```

6.2 Revisions

The process of model checking is an iterative process, with many refinement steps before eventually reaching a final working model. As mentioned in the background, a refinement step for model checking can yield that either the models or the properties need to be changed. This section will show some examples which made refinements to the model.

The Atomic Requirement

A big concern when modeling concurrency is the uncertainty of interleaving statements. To address such issues, one can use a modelling tool called **atomic** statements, which specify that several statements should be executed "at once" (without interleaving). These should be used with careful consideration, to not change the nature of what is sought to model. Our project experienced a 'state space explosions' (mentioned in a previous chapter). To reduce the state space, we introduced atomic statements at two steps in our models. One example is shown below:

Listing 6.4: Atomic statement

```
Waiting:
            atomic {
              if
              :: out ? continue -> goto Idle;
              :: out ? stop ->
              fi;
            }
```

This section also had problems with interleaving statements, it was the step where a node received the information to whether stop or continue collecting data. This change was due to that Promela uses communication channels to handle message passing and previously a shared buffered channel had been used. Though from verification with model checking, the amount of states rapidly increased with a buffered channel. So instead the project had to modify the models to use a unique channel between the server and each node. The atomic statement here handled the logic that a node that had received a message didn't wait, but instead immediately acted on the information.

Introduction of States

As can be seen in the initial model, this version handled the verification by using global variables to monitor certain values. An example is the variable `nodesDone` which represented on how many nodes were currently actively collecting data. This approach worked initially, but made the models lack certain information. Specifically for the liveness property to know whether a specific node had received data or not. This caused the introduction of states in the models instead. Which also made the models easier compare to their sought FSAs. This resulted in a large revision, as previously the procedures' statements were written to be computationaly efficient and easy to read. The largest change from this can be seen in **??**, where polling was used instead of waiting.

The Network Actor

When the project began modelling what we called *decentralized decision making*, meaning that we modelled e.g. a system where nodes had a processing unit and could process data rather than just forwarding it. We attempted solutions with intermediate channel between the nodes, but as we previously had explored when introducing `atomic`-statements, we were unable to verify it since the state space grew too large with increasing number of collection nodes (a state space explosion). Instead we decided to introduce an additional Actor in the system, called the *Network Actor*. This Actor would listen on a channel `broadcast`, and upon reaching a `stop`-message it would broadcast to all nodes in the system to stop collecting. This was a simplification which abstracted our models from what we in reality tried to model. Therefore we introduced an `atomic`-statement here aswell, which made a "node's request to broadcast" logically the same as what we sought.

Listing 6.5: Network Actor

```
active proctype Network() {
  int i=0;
Idle:
  if
  :: atomic {
      broadChan ? stop ->
      for(i : 0 .. (NUM_NODES-1)) {
        networkChan[i] ! stop;
      }
    }
  fi;
}
```

The Liveness Relaxation

The liveness property for the system demanded several revisions. As defined, a liveness property should express that *eventually something good happens*. For this project this was it was initially stated as:

eventually the server stops collecting or a stop-decision is never sent.

Intuitively this seemed correct we were unable to verify this property in SPIN, model checking yielded that there existed a case where this property didn't hold. So after revising this formula several times we instead sought a liveness property stating the following;

if over-collection is about to (or would) occur, nodes should stop collection.

In order to model this, we had to know when over-collection is occuring. For our system that meant when `bigData` was metered by a node, this lead to a relaxation of the FSA for the environment. If it was ever metered, the environment actor would enter a new state so that our verification could check this. This change abstracted our models, but the language of the automata still remained the same which was important to allow the change.

Listing 6.6: Environment code

```
active proctype Env() {

Idle:
        if
        :: atomic {
            envChan ? meter ->
              if // random outcome
              :: envChan ! bigData; goto Idle_bigData;
              :: envChan ! smallData;
              fi;
            goto Idle;
          }
        fi;

Idle_bigData:
        if
        :: atomic {
            envChan ? meter ->
              if // random outcome
              :: envChan ! bigData;
              :: envChan ! smallData;
              fi;
            goto Idle_bigData;
          }
        fi;
}
```

7 Discussion

This section presents a discussion and suggestions for future work.

7.1 Discussion of future work

During the theoretical work of the project we researched using decision procedures for defining Over-Collection in Wireless Sensor Networks . As we didn't have time for the extended models, nor the implementation, the development of decision procedures didn't become relevant. A project could instead focus on chosing a good formal system and then developing decision procedures that decides if Over-Collection is occuring in a system. With a general solution or a clear method for reaching this aim, this could in turn help:

- Formulating a tool to be used for designing systems that collects data, to prevent Over-Collection.
- or, A solution that could be attached to existing networks, to prevent Over-Collection.

Which would in turn yield better privacy and Data Minimization.

8
Conclusion

In this thesis, we investigated over-collection in Wireless Sensor Network . We kept ourselves within the limitations and We divised a terminology for our system, with e.g. *Actors* and *Behavior Models*, for our analysis of the different entities in a Wireless Sensor Network . With this terminology we described how each entity in the system should act upon received information.
We then constructed an abstract model from this, using Promela as a tool, of a Wireless Sensor Network collecting data and that used Data Minimization to prevent over-collection. We then specified safety correctness and liveness properties for our system in Linear Time Logic, using Model Checking in SPIN to verify that our system was working as intended.
In our analysis we found further variations for Wireless Sensor Networks , as mentioned in Section 5.3.2, that introduced new aspects for our analysis into Data Minimization in Wireless Sensor Networks . This analysis led to further modelling and refinement of our properties. The designs of said models can be seen in Appendix 1, also a more thorough description into one of the models can be found in the Design in Section 6.1.
The project achieved a good set of models, to use as a foundation for developing the extended models sought in Phase 2. The modelling in Phase 1 took longer than expected which led to that Phase 2 were never reached. We believe that the analysis in Phase 2 would have yielded very interesting results into Data Minimization. Which could help addressing problems that arises with Over-Collection.
To summarize, modelling Wireless Sensor Networks and using Model Checking to formally verifying their behaviour is a good approach. Modelling several systems behaving and acting differently and verifying them with the same properties was more challenging than expected. Even though this project did not implement a solution from the models, it's findings can still be useful for similar work into Model Checking and analysation of Data Minimization.

Bibliography

[1] D. Miorandi, S. Sicari, F. De Pellegrini, and I. Chlamtac, “Internet of things: Vision, applications and research challenges,” *Ad Hoc Networks*, vol. 10, no. 7, pp. 1497–1516, 2012.

[2] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “Wireless sensor networks: a survey,” *Computer networks*, vol. 38, no. 4, pp. 393–422, 2002.

[3] I. F. Akyildiz, T. Melodia, and K. R. Chowdhury, “A survey on wireless multimedia sensor networks,” *Computer networks*, vol. 51, no. 4, pp. 921–960, 2007.

[4] R. Min, M. Bhardwaj, S.-H. Cho, E. Shih, A. Sinha, A. Wang, and A. Chandrakasan, “Low-power wireless sensor networks,” in *VLSI Design, 2001. Fourteenth International Conference on*, pp. 205–210, IEEE, 2001.

[5] K. Sohrabi, J. Gao, V. Ailawadhi, and G. J. Pottie, “Protocols for self-organization of a wireless sensor network,” *IEEE personal communications*, vol. 7, no. 5, pp. 16–27, 2000.

[6] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar, “Next century challenges: Scalable coordination in sensor networks,” in *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pp. 263–270, ACM, 1999.

[7] A. Elahi, “City seeks input on privacy policy for array of things sensor network,” June 2016. [Online; posted 10-June-2016].

[8] A. Baggio, “Wireless sensor networks in precision agriculture,” in *ACM Workshop on Real-World Wireless Sensor Networks (REALWSN 2005), Stockholm, Sweden*, Citeseer, 2005.

[9] T. C. Collier, A. N. Kirschel, and C. E. Taylor, “Acoustic localization of antbirds in a mexican rainforest using a wireless sensor network,” *The Journal of the Acoustical Society of America*, vol. 128, no. 1, pp. 182–189, 2010.

[10] K. K. Khedo, R. Perseedoss, A. Mungur, *et al.*, “A wireless sensor network air pollution monitoring system,” *arXiv preprint arXiv:1005.1737*, 2010.

[11] S. Vardhan, M. Wilczynski, G. Portie, and W. J. Kaiser, “Wireless integrated network sensors (wins): distributed in situ sensing for mission and flight systems,” in *Aerospace Conference Proceedings, 2000 IEEE*, vol. 7, pp. 459–463, IEEE, 2000.

[12] W. M. Merrill, F. Newberg, K. Sohrabi, W. Kaiser, and G. Pottie, “Collaborative networking requirements for unattended ground sensor systems,” in *Aerospace Conference, 2003. Proceedings. 2003 IEEE*, vol. 5, pp. 5_2153–5_2165, IEEE, 2003.

[13] A. Darwish and A. E. Hassanien, "Wearable and implantable wireless sensor network solutions for healthcare monitoring," *Sensors*, vol. 11, no. 6, pp. 5561–5595, 2011.

[14] E. Ertin, N. Stohs, S. Kumar, A. Raij, M. al'Absi, and S. Shah, "Autosense: unobtrusively wearable sensor suite for inferring the onset, causality, and consequences of stress in the field," in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pp. 274–287, ACM, 2011.

[15] H. Chan and A. Perrig, "Security and privacy in sensor networks," *computer*, vol. 36, no. 10, pp. 103–105, 2003.

[16] M. Al Ameen, J. Liu, and K. Kwak, "Security and privacy issues in wireless sensor networks for healthcare applications," *Journal of medical systems*, vol. 36, no. 1, pp. 93–101, 2012.

[17] E. D. P. Supervisor, "Regulation (ec) no 45/2001," December 2000.

[18] G. Danezis, J. Domingo-Ferrer, M. Hansen, J.-H. Hoepman, D. L. Metayer, R. Tirtea, and S. Schiffner, "Privacy and data protection by design-from policy to engineering," *arXiv preprint arXiv:1501.03726*, 2015.

[19] A. Pfitzmann and M. Hansen, "A terminology for talking about privacy by data minimization: Anonymity, unlinkability, undetectability, unobservability, pseudonymity, and identity management," 2010.

[20] C. Ramamoorthy and G. S. Ho, "Performance evaluation of asynchronous concurrent systems using petri nets," *IEEE Transactions on software Engineering*, no. 5, pp. 440–449, 1980.

[21] J. C. Baeten, "A brief history of process algebra," *Theoretical Computer Science*, vol. 335, no. 2-3, pp. 131–146, 2005.

[22] B. C. Pierce, "Foundational calculi for programming languages.," *The Computer Science and Engineering Handbook*, vol. 1997, pp. 2190–2207, 1997.

[23] S. Chong and A. C. Myers, "Language-based information erasure," in *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pp. 241–254, IEEE, 2005.

[24] M. Hansen, P. Berlich, J. Camenisch, S. Clauß, A. Pfitzmann, and M. Waidner, "Privacy-enhancing identity management," *Information security technical report*, vol. 9, no. 1, pp. 35–44, 2004.

[25] P. Bjesse, "What is formal verification?," *ACM SIGDA Newsletter*, vol. 35, no. 24, p. 1, 2005.

[26] E. M. Clarke, O. Grumberg, and D. Peled, *Model checking.* MIT press, 1999.

[27] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, p. 279, 1997.

[28] D. Peled, T. Wilke, and P. Wolper, "An algorithmic approach for checking closure properties of ω-regular languages," *CONCUR'96: Concurrency Theory*, pp. 596–610, 1996.

[29] D. Kroening and O. Strichman, *Decision Procedures.* Springer, 2008.

[30] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *International Conference on Computer Aided Verification*, pp. 519–531, Springer, 2007.

[31] B. Dutertre and L. De Moura, "A fast linear-arithmetic solver for dpll (t)," in *International Conference on Computer Aided Verification*, pp. 81–94, Springer, 2006.

[32] C. Barrett, D. Dill, and J. Levitt, "Validity checking for combinations of theories with equality," in *Formal Methods In Computer-Aided Design*, pp. 187–201, Springer, 1996.

[33] C. Barrett and S. Berezin, "Cvc lite: A new implementation of the cooperating validity checker," in *International Conference on Computer Aided Verification*, pp. 515–518, Springer, 2004.

[34] S. K. Lahiri and S. A. Seshia, "The uclid decision procedure," in *International Conference on Computer Aided Verification*, pp. 475–478, Springer, 2004.

[35] J. Newsome, D. Brumley, J. Franklin, and D. Song, "Replayer: Automatic protocol replay by binary analysis," in *Proceedings of the 13th ACM conference on Computer and communications security*, pp. 311–321, ACM, 2006.

[36] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.

A
Appendix - Model Designs

A.1 Models

Listing A.1: centralized_conjunctive_centralized.pml

```
#define NUM_NODES
#define node_send         Node@Waiting
#define server_dC         Server@Stopping
#define node_done         Node@DoneColl
#define server_ans        Server@Answering
#define server_stop       Server@Stopping

mtype = {meter, bigData, smallData, continue, stop, ack};

ltl correctness { always (server_dC implies (eventually node_done)) }

ltl liveness { (not node_notify until bigData_metered) }


chan envChan = [0] of {mtype};
chan servChan[NUM_NODES] = [1] of {mtype};

init {
  atomic {
    int i;
    for (i : 0 .. (NUM_NODES-1)) {
      run Node(servChan[i]);
    }
  }
}


active proctype Env() {

Idle:
      if
      :: atomic {
          envChan ? meter ->
           if // random outcome
           :: envChan ! bigData; goto Idle_bigData;
           :: envChan ! smallData;
           fi;
          goto Idle;
        }
        fi;

Idle_bigData:
        if
        :: atomic {
             envChan ? meter ->
               if // random outcome
               :: envChan ! bigData;
               :: envChan ! smallData;
               fi;
             goto Idle_bigData;
           }
        fi;
}

active proctype Server() {

bool data[3];
data[0]=false;
data[1]=false;
data[2]=false;
int index=0;
int i=0,j=0;

Idle_Answering:
          if
          :: nempty(servChan[i]) ->
               j=i;
               i=(i+1)%NUM_NODES;
               goto Answering;
          :: empty(servChan[i]) ->
               i=(i+1)%NUM_NODES;
               goto Idle_Answering;
          fi;

Answering: // updating index will only be relevant here
          if
          :: servChan[j] ? smallData ->
               data[index]=false;
               index=index+1; // index++
               servChan[j] ! continue;
               goto Idle_Answering;
          :: servChan[j] ? bigData ->
               if
               :: (data[0] || data[1] || data[2]) ->
                 servChan ! stop;
                 goto Idle_Stopping;
               :: else ->
                 data[index]=true;
                 index=index+1; // index++
               fi;

          fi;

Idle_Stopping:
          if
        :: nempty(servChan[i]) ->
            j=i;
            i=(i+1)%NUM_NODES;
            goto Stopping;
        :: empty(servChan[i]) ->
            i=(i+1)%NUM_NODES;
            goto Idle_Stopping;
        fi;

Stopping:
        if
        :: servChan[j] ? smallData ->
            servChan[j] ! stop;
            goto Idle_Stopping;
        :: servChan[j] ? bigData ->
            servChan[j] ! stop;
            goto Idle_Stopping;
        fi;
}

proctype Node(chan out) {
Idle:
        envChan ! meter;
        if
        :: envChan ? bigData ->
           out ! bigData;
           goto Waiting;
        :: envChan ? smallData ->
           out ! smallData;
           goto Waiting;
        fi;
Waiting:
        atomic {
          if
          :: out ? continue -> goto Idle;
          :: out ? stop ->
          fi;
        }
DoneColl:
}
```

A.2 Initial Model

Listing A.2: centralized_decision_multiple_nodes_8jul

```

#define N   3

mtype = {ack, stop, collect, send};
int nodesDone=0;

chan ntoS = [2] of {mtype, int, int};
chan stoN = [2] of {mtype, int};
chan etoN = [2] of {mtype, int, int};
chan ntoE = [2] of {mtype, int};
```

```

proctype Environment(chan in, out) {
  int r=0; // r is the data from the environment
  int id;
  printf("Environment starting up.\n");
  do
  :: (nodesDone == N) ->
     break;
  :: (nodesDone<N) && (nempty(in)) ->
     do
     :: r=1;  break;
     :: r=2;  break;
     :: r=3;  break;
     :: r=4;  break;
     :: r=5;  break;
     :: r=6;  break;
     :: r=7;  break;
     :: r=8;  break;
     :: r=9;  break;
     :: r=10; break;
     :: r=11; break;
     od;

     in ? collect(id);

     out ! ack(id, r);

  od;
  printf("Environment done, shutting down.\n");
}

/* inS = incoming from server
   outS = output to server
*/
proctype Node(chan inS, outS, inE, outE) {
  int data;
  int id=_pid;
  mtype msg;
  int c = 1;

  printf("Node %d starting up.\n", id);

  do
  :: outE ! collect(id); // request info from environment
     inE ? ack(id, data); // receive - || -
     outS ! send(id, data);
     inS ? msg,id;
     if
     :: (msg == stop) -> break;
     :: (msg == ack) ->  c=c+1;
     :: else ->
     fi;
  od;

  nodesDone = nodesDone+1;
  printf("Node %d done, shutting down. Did this %d times.\n", id, c);
}


proctype Server(chan in, out) {
  int data;
  int id;
  printf("Server starting up.\n");
  do
    :: (nodesDone == N) ->
        break;
    :: (nodesDone<N) && (nempty(in)) ->
        in ? send(id, data);
        if
        :: (data > 9) ->
            out ! stop(id);
        :: else ->
            out ! ack(id);
        fi;
        printf("Collected %d from Node %d.\n", data, id);
  od;
  printf("Server done, shutting down.\n");
}

init {
  run Server(ntoS, stoN);
  run Node(stoN, ntoS, etoN, ntoE);
  run Node(stoN, ntoS, etoN, ntoE);
  run Node(stoN, ntoS, etoN, ntoE);
  run Environment(ntoE, etoN);
}
```