An Approach to Semi-Automatic Code Generation for the TinyOS Platform using Coloured Petri Nets

Master's Thesis

Vegard Veiset
Software Engineering
University of Bergen & Bergen University College
Norway

Supervised by
Lars M. Kristensen
Bergen University College

May 2013

Abstract

TinyOS is a widely used platform for the development of network embedded systems such as distributed sensor networks, targeting low-powered wireless devices. We present a software engineering approach where Coloured Petri Net (CPN) models are used as the starting point for developing protocol software for the TinyOS platform. The approach consists of a five step refinement process taking a platform-independent CPN model and gradually refining it to match the structure of the target platform, ending up with a refined model that enables automatic code generation. We have used a case study of the IETF Roll Routing protocol to evaluate our approach.

Contents

Chapter 1

Introduction

Models of software can give insight into an idea, a concept or a design. It can help us to get a better understanding of a software system and highlight details that were previously unnoticed. When a software system has been represented as a model, we would like to leverage the time invested in creating the model to automatically generate implementation code for a concrete platform. By doing this, we can eliminate the number of human errors that would occur if the implementation had to be done manually, as well as save time that has already been invested in creating the model.

One drawback of implementing an abstract model instead of a concrete implementation of a software system, is that we do not have an implementation that can be executed on the target platform. In this thesis, we look closer at how we can use an abstract model to generate implementation code for a concrete platform, and which refinements are needed to obtain a model from which we can generate platform specific code.

Network protocols are concurrent, non-deterministic, and as a result they are often complex. When implementing protocols, rigorous testing is needed to ensure the correctness of the protocol implementation. By specifying the protocol using abstract models, we can simulate the behaviour and eliminate a number of errors that we might not have discovered otherwise. From this perspective, we investigate in this thesis the Roll Protocol[10], a network protocol specified by the Internet Engineering Task Force (IETF). The Roll Protocol (RPL) is an IPV6 routing protocol for low-power and lossy networks, and is well suited for nodes in wireless sensor networks. We have chosen to use the TinyOS platform as a case study for code generation. TinyOS is a platform targeting low powered devices.

The specific focus of this thesis is on how we can refine and use pragmatics[18] to annotate Coloured Petri Net[11] (CPN) network protocol models with additional information that will enable us to generate nesC code for the TinyOS platform[15].

1.1 Coloured Petri Nets and Code Generation

Coloured Petri Nets (CPN) offer a way of creating abstract models that can be used to represent concurrency, non-deterministic behaviour and communication in software systems. Because of the abstract nature of CPN models, deriving code automatically from the models is challenging due to the gap between an abstract and platform independent model and code that can be executed on a specific platform. CPN Tools is a tool for modelling concurrent and non-deterministic behaviour and analyzing CPN models[11].

A CPN model of a software system can represent complex behaviour which it can be challenging to reason about. We can simulate the behaviour and make reasonable estimates about the behaviour of a model and generate code based on that. This has been done in the simulation based code generation approach of [13, 17]. Another approach is to use structure-based automatic code generation[8] based on Process-Partitioned CPNs, an extension of CPN. There are projects that generate CPN models based on software implementations. An example of this is the `nesc2cpn`[7] program that takes a TinyOS application and generates a CPN model that is used for estimating the power consumption of the application.

1.2 Model Refinement

Generic abstract models are hard to relate to concrete implementations for a target platform. By adding details and restructuring (refining) the generic model, we can adapt it to match the target platform, and this enables us to relate the abstract model to the specific platform.

The refinement of the CPN model in this thesis is largely based on the use of pragmatics[18] to annotate the model with additional explicit information. The reader is introduced to the five steps of CPN model refinement for TinyOS network protocol applications that we have created.

We have implemented a CPN model representation of the Roll Protocol. This implementation was created prior to having knowledge about the target platform. This is done to keep the original model as platform-independent as possible. Having a platform-independent model increases our confidence in that the CPN

model refinement steps we have created are generic and can be applied for other network protocols.

1.3 Thesis Goal and Results

The goal of this thesis is to further explore the possibilities of getting from an abstract CPN model into implementation code for a target platform, while ensuring readability of the generated code. The TinyOS platform will be the primary frame of reference to demonstrate the applicability of the research conducted in this thesis. We use a case study of the Roll Protocol[10], a protocol for routing over low-powered and lossy networks, to explore the characteristics of structure-based automatic code generation. The thesis investigates several related research questions, and the research questions are investigated by means of software prototypes. The research questions are as follows:

- How can we transform CPN models into platform specific code for the TinyOS platform?
- How do we refine the CPN model sufficiently to use it for code generation?
- Are pragmatics suitable for the model refinement process?
- What general steps are required in the refinement process?

Figure 1.1 shows the process of going from a specification to implementation of executable code. The Roll Protocol specification is modelled using CPNs. Using CPN Tools, we are able to simulate and verify the behaviour of the model. To generate code for the target platform, the model-to-code transformation (code generator) requires a CPN model sufficiently refined, and annotated with pragmatics giving additional detail about the target platform.

The first step is to create a CPN model representing the behaviour of the Roll Protocol. The second step is to refine the CPN model by adding more detail to it, and this is done by annotating the CPN model and by changing the structure of the model to better match the target platform. We refine the model to the point where we have got enough detail to be able to generate platform specific code from it. We use Access/CPN[21] to load CPN models created with CPN Tools.

By going through the process of refining the CPN Roll Protocol model, we have derived five refinement steps that can be used to translate generic CPN network protocol models into CPN models from which we can generate TinyOS applications. We show how we are able to generate the application structure and the outline of the behaviour of TinyOS commands and events. The source code generated by

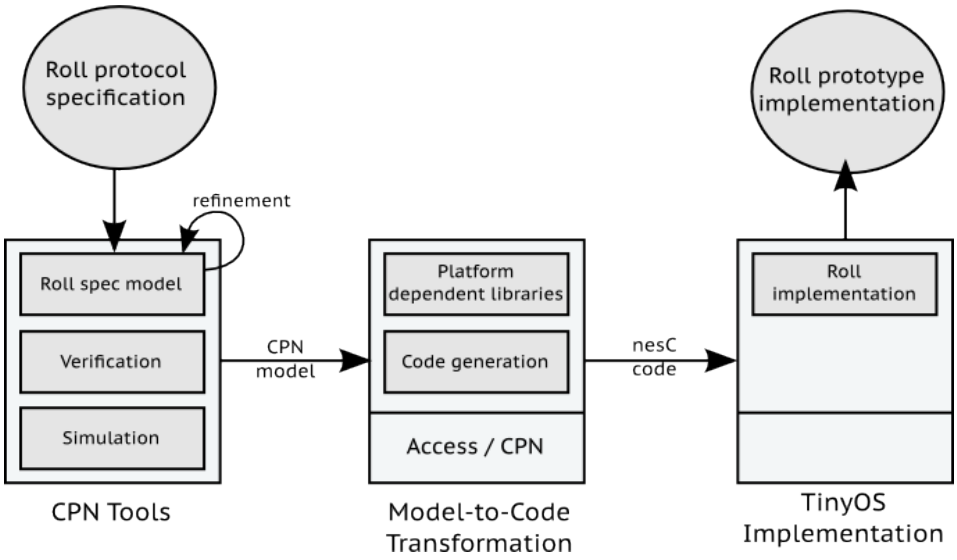


Figure 1.1: Research approach using prototypes

the code generator does, however, require some minor manual implementation to be runnable. Hence, what we have developed is a semi-automatic approach to code generation for the TinyOS platform.

1.4 Thesis Outline

The reader is assumed to have basic knowledge of the C programming language, the Standard ML programming language, and know the core concepts of Petri Nets. Below we briefly summarize the content of the individual chapters that constitute this thesis.

Chapter 2 - Coloured Petri Nets and the Roll Protocol introduces CPN Tools, and how it has been used for modelling the Roll Protocol[10]. By describing the process of creating the CPN Roll Protocol model, we introduce the concepts needed to understand Coloured Petri Nets and the Roll Protocol.

Chapter 3 - The nesC Programming Language and the TinyOS Platform introduces the key concepts of the nesC programming language and the TinyOS platform. To introduce the nesC programming language and TinyOS platform, we show how selected parts of the Roll Protocol can be implemented in nesC.

Chapter 4 - CPN Model Refinement describes how the CPN Roll Protocol model has been refined. The refinement has been done by adding TinyOS

platform specific details to the model. The reader will be given an overview and motivation for each of the steps in the refinement.

Chapter 5 - Code Generation introduces the technical details on how to generate code for the TinyOS platform based on a CPN model refined according to the steps introduced in Chapter 4. We introduce how the code generator works, and the result of each step in the code generation.

Chapter 6 - Application to Roll shows the necessary manual steps needed to go from the generated code and to a running network protocol application.

Chapter 7 - Conclusions and Future Work discusses the results of the refinement, the generated code and the process of going from an abstract CPN model to a concrete implementation of a network protocol for the TinyOS platform. We outline possible extensions and improvement to our code generator, and provide direction for future work.

The source code for the code generator, related documentation, revisions of the CPN models during the refinement process, binaries, and code samples can be found online at `http://veiset.org/master/`.

Chapter 2

Coloured Petri Nets and the Roll Protocol

This chapter introduces core concept of Coloured Petri Nets, wireless sensor networks, and the Roll Protocol. We use a CPN model representation of the Roll Protocol to introduce the reader to the operation of the Roll Protocol. When introducing the CPN model of the Roll Protocol, we introduce the core concepts needed to understand how CPNs works, and how CPN Tools is used in the development of abstract and platform independent models.

2.1 Roll protocol

The Roll Protocol[10], also know as *RPL*, is an IPv6 routing protocol for low-power and lossy networks. Low-power and lossy networks are networks where high rates of packet loss and low transfer rates are expected[1]. Lossy networks often constitute environments where the processing power, memory and power consumptions have constraints.

We find lossy networks in areas such as distributed sensor networks. Sensor networks can be used in hostile environments where it would be hard to install and maintain physical wiring between nodes. An example of this is the Golden Gate Bridge safety high-speed accelerometers[12]. Another appliance of sensor networks are in data center facilities. Data centers use a lot of power on cooling and by attaching sensor nodes to the machine racks it has been shown[16] to be possible to dynamically adapt the cooling needed to reduce the overall power consumption of the data center.

The nodes in the network might not always have the correct, and most up-to-date information about the network. This is because constantly updating the

routing information is expensive both in terms of network load and node resource consumption.

An instance of the Roll Routing Protocol (RPL) organizes nodes in a network as a directed acyclic graph, and consist of one or more root nodes which are acting as sinks for the network. The Roll Protocol discovers links and selects peers sparingly.

2.1.1 Overview

The Roll routing protocol uses *Destination-Oriented Directed Acyclic Graphs* (*DODAGs*[10]) for building network routes consisting of nodes (devices), and uses IPV6 messages[9] (ICMPv6) to communicate between the devices. There are five different types of control messages that are defined by the Roll Protocol specification[10]. These are used for network discovery, propagating information, sending data, inconsistency checks, and packet encryption.

An instance of the routing protocol consist of one or more Destination-Oriented Directed Acyclic Graphs. A *DODAG* typically only has one root node, and in the case that there are multiple root nodes defined, it is assumed that they all are connected to a common backbone network. A *DODAG* is organized as a directed graph consisting of children and parent nodes. A child typically only has one parent, while a parent-node can have multiple children. The traffic flow in a *DODAG* can go in two directions: unicasted upwards in the graph from child to parent (towards the root node), or downwards by being broadcasted to all child nodes.

A *DODAG* is uniquely identified by the combination of a *DODAGID* (which is a unique number identifying the *DODAG* root-node), a *RPLInstanceID* (which is an unique ID identifying the network), and a *DODAGVersionNumber*. The *DODAGVersionNumber* is the current iteration number of the *DODAG*. The *DODAG* contains a finite number of nodes that have a rank associated with them.

The *DODAGID*, *RPLInstanceID*, *DODAGVersionNumber* and the *rank* of a node makes up the RPL identifiers. The RPL identifiers are used to identify, manage, and maintain a topology.

Figure 2.1 shows how a network topology can be represented as a *DODAG*. The nodes in the network will attempt to find the best suited parent based on an objective function. The Network_as_DODAG (Figure 2.1 - middle) shows the route of the packets going towards the root node. The objective function describes how the nodes should choose their parents based on attributes such *rank*, *DODAGVersionNumber* and *DISTANCE*.

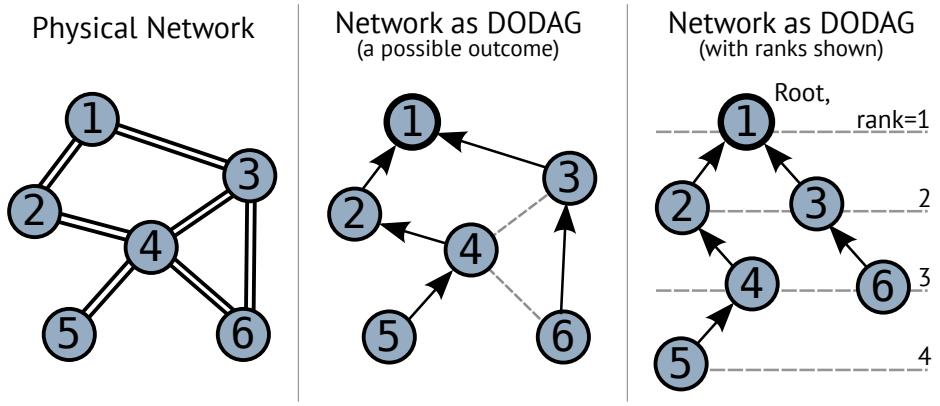


Figure 2.1: Physical network topology represented as a DODAG

Figure 2.2 shows how nodes in a network can exchange packets to join and form a *DODAG*. The scenario uses the network in Figure 2.1, with one root node (root1) and five non-root nodes (node 2-6). The root-node (root1) is a preconfigured node in the network that is acting as a sink for a common network. The left side of the Figure 2.2 shows the exchange of network packets, while the right side shows the current *DODAG* representation of the connected nodes.

The nodes will send discovery requests (DIS in Figure 2.2) with their current *rank* and *DODAGVersionNumber* in attempt to find a *DODAG* to join. When a node is part of an RPL instance (has joined a *DODAG*) it will respond to incoming discovery requests with a discovery response (DIO). The node will then, based on the incoming responses, pick the most suited parent according to an objective function. The objective function zero[20] (*OF0*) is used to calculate the most favourable parent, and will always favour the parent with the highest *DODAGVersionNumber*. The *OF0* will pick the one with the lowest *rank* within that version of the *DODAG*. When a node gets multiple responses with suited parents, the node will choose the parent as described by the objective function. The node will evaluate the response regardless of whether it already has a parent or not, and choose the optimal one. This is illustrated in Figure 2.2 by node6.

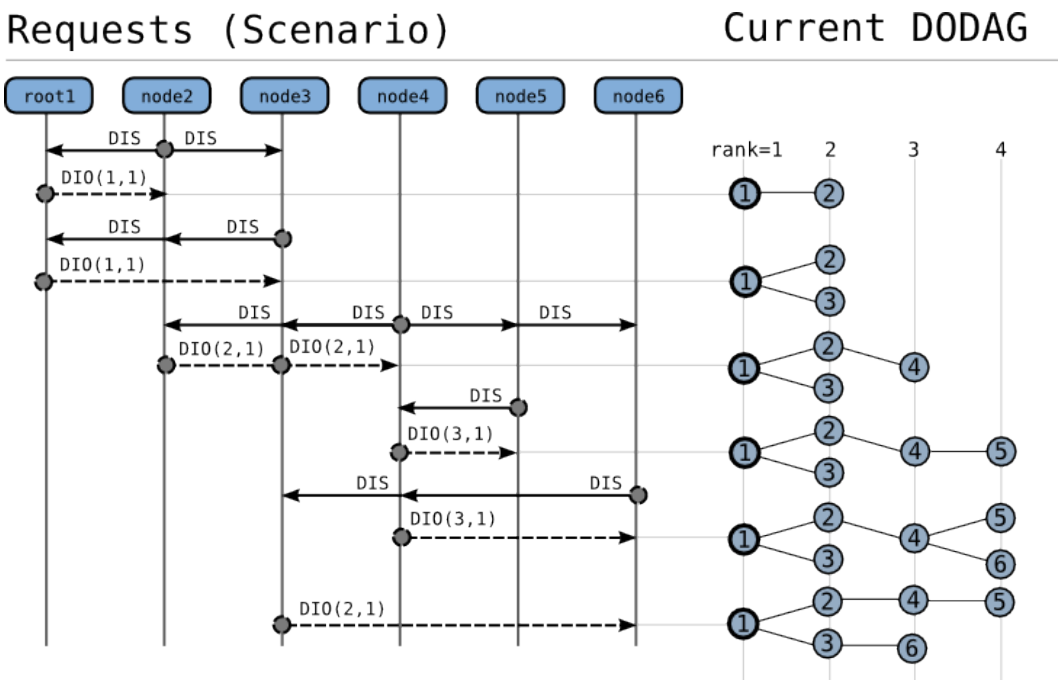


Figure 2.2: Nodes forming a new instance of a DODAG

2.2 The CPN Roll Protocol Model

We create CPN models by using transition, places, arcs and inscriptions. Transitions are used to add and remove tokes from places, and places contains the data. The arcs describes the data types and constraints on the data that will be added or removed from places. Inscriptions are used to define functions and expressions (written in CPN ML) that are evaluated to a multiset or a single element. Our model consists of a hierarchy of modules.

The CPN Roll model is based on the Roll Protocol as specified in *RFC6550*[10]. The model specifies how nodes in a network obtain configuration parameters (*DODAGVersionNumber* and *Rank*) from neighbouring nodes and how the nodes choose their parents based on that information. In the model, the nodes choose their parents according to the *ObjectiveFunctionZero*[20]. Furthermore, the CPN model allows nodes to discover that their parents have disconnected. This is done by sending destination advertisement (DAO) packets explicitly asking for acknowledgements. Our model does not include the consistency checks and security aspects of *RFC6550*.

The top-level of the CPN-model, is shown in Figure 2.3. We are using substitution transitions to structure large models into smaller parts. The substitution transition RollProtocol in Figure 2.3 is handling the logic of the Roll Protocol, while the substitution transition LinkLayer represents the network link layer of our model.

A hierarchical CPN model can always be represented as a flat model, but for readability and maintainability we have chosen to organize our model in an hierarchy. We have two main modules, one module representing the network and one representing the Roll Protocol. To get places to connect and interact with submodules (substitution transitions) we use socket places. Socket places can be inputs, outputs or both and are connected to substitution transitions.

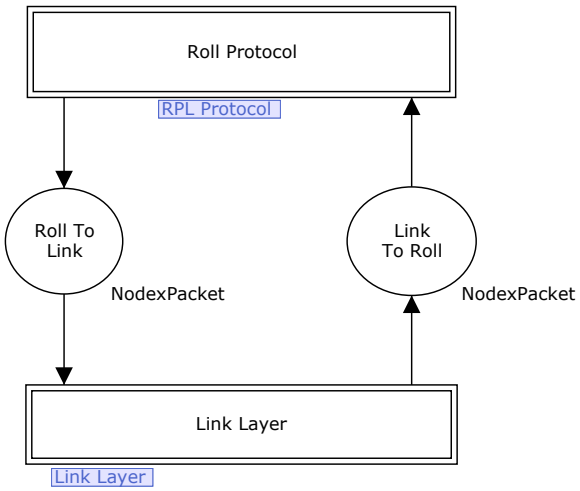


Figure 2.3: Overview of the CPN-Roll model

Figure 2.4 shows the hierarchy of our model. The hierarchy consist of eight modules. The main module, Main, containing two submodules, one for the link layer and one for the protocol as shown in Figure 2.3. The RollProtocol module contains the behaviour of the routing protocol and has four submodules containing logic for: discovering and joining (DISDIO), sending destination advertisements (DAO), handling acknowledgements (DAOACK), and initial joining and local configuration. The module LinkLayer representing the link layer, is responsible for transmitting data between the nodes in the Roll Protocol. The link layer has a submodule that contains logic for changing the network topology.

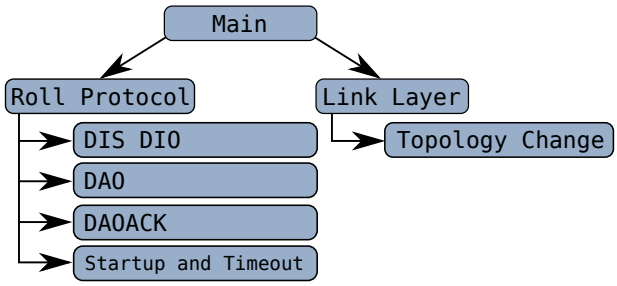


Figure 2.4: CPN-Roll Protocol Module Hierarchy

2.2.1 Roll Protocol Module

The Roll Protocol module consists of four main modules: Discovering and joining a *DODAG* (DIS DIO), sending packets containing payload (DAO), acknowledging packets containing payloads (DAOACK), and node configuration and behaviour (Startup_and_Timeout).

Figure 2.5 shows the four substitution transitions that make up the CPN Roll Protocol module. The place DodagState represents the information each node currently has about the *DODAG* and in which state the node is in. LinkToRoll is the place for incoming packets from the link layer, and RollToLink are the packets being sent to the link layer, i.e, outgoing packets from the node.

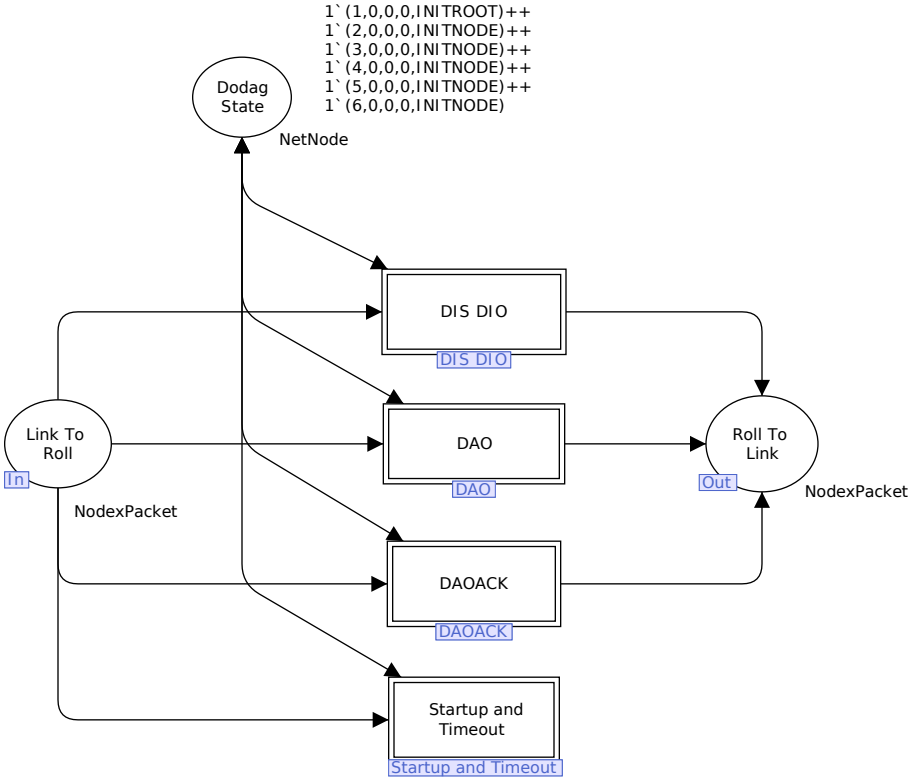


Figure 2.5: CPN-Roll Protocol Module

The CPN model describes three main active states that a node can be in:

- `JOINING` - Trying to find and connect to an existing *DODAG*
- `JOINED` - Currently connected to a *DODAG*
- `WAITING` - Connected to a *DODAG*, and waiting for an acknowledgement

We have two types of nodes: root nodes and non-root nodes. Both non-root nodes and root-nodes will start in a booting-state, *INITNODE* and *INITROOT*

respectively. A root node will not attempt to discover or rejoin a *DODAG* as the root node is a preconfigured entity in the network and will be in the state *ROOTJOINED* until disconnected. Being a root-node allows the node to increase the *DODAGVersionNumber*. Non-root nodes will start in the state of *JOINING* and this will enable them to probe the network to find information about the *DODAG*, and thus allowing them to join.

We have created an enumerated colour set `STATE` (Figure 2.6) listing the states that the nodes can be in. The `with` keyword is used to specify a list of allowed elements that can be used as values in the colour set `STATE`. The `STATE` colour set is defined as follows:

```
1 colset STATE = with INITROOT | INITNODE | ROOTJOINED
2                  | JOINED | JOINING | WAITING;
```

Figure 2.6: CPN colour set representing a state in the Roll Protocol

Furthermore, Figure 2.7 shows the defined colour sets for nodes, rank of nodes, and the colour set for the *DODAG* version number.

```
1 colset Nodes     = int with 0..N;
2 colset Rank      = int;
3 colset DodagVerNum = int;
```

Figure 2.7: CPN color sets for Node, Rank and DodagVersionNumber

A node in our model is represented as a NetNode (see below) and has a unique ID. The `NetNode` is a product of five colsets, in order, representing: An identification number (`Nodes`), a rank (`Rank`), a version number (`DodagVerNum`), a parent (`Nodes`) and a state (`STATE`).

```
colset NetNode = product Nodes * Rank * DodagVerNum * Nodes * STATE;
```

In addition to the `STATE` color set, we have created a color set for packet types which is a union of the packet types of the Roll Protocol. Using pattern matching in arc expressions allows us to enable transitions only when nodes are in certain states and/or receiving specific types of packets.

We have defined a color set `Packet` to be a union of a destination and a packet type. The packet type is a representation of each of the different types of packets we have chosen to include in our CPN model: *DIS*, *DIO*, *DAO* and *DAO-ACK*.

The `Packet` colour set (Line 5, Fig. 2.8) contains information about the packet and the destination. The destination can either be a unicast to a single neighbour

(`DEST(n)`) or a multicast to all the neighbours (`DEST(ALL)`). The `Nodes` colour set contains a single ID for a node in the network, and in the colour set `NodexPacket` this represents the source node of the packet.

```
1 colset PacketType = union DIS
2                   + DAO:DAOpack
3                   + DAOACK:DAOACKpack
4                   + DIO:DIOpack;
5 colset Packet = product Dest * PacketType;
6 colset Dest = union ALL + DEST:Nodes;
```

Figure 2.8: CPN color set for a generic RPL network packet

The *DIS* packet contains no extra information relevant to the Roll specification and is a constant. *DIO* and *DAO-ACK* packets contains information about the rank and version number of the sender. The *DAO* packets (`DAOpack`) in addition to this also contains `Data` and a option field `Options`. The option field is used to inform the receiver whether it should respond with a *DOA-ACK* or not.

```
1 colset DIOpack = product Rank * DodagVerNum;
2 colset DAOpack = product Rank * DodagVerNum * Data * Options;
3 colset DAOACKpack = product Rank * DodagVerNum;
```

Figure 2.9: CPN color sets for the different RPL packets

We are using a colour set of type `NodexPacket` to describe packets going over the network. `NodexPacket` is a product of the colour sets `Nodes` and `Packet`. And is defined as:

```
colset NodexPacket = product Nodes * Packet;
```

The DIS DIO module

The DIS DIO module contains logic for obtaining information about a *DODAG* as well as the logic for joining a *DODAG* based on an objective function.

The node wanting to join the network (`state=JOINING`) will send a DODAG Information Solicitation (*DIS*) to obtain information about nearby *DODAG* instances. This is done by probing the neighbouring nodes in the physical network. When a node is part of a *DODAG* (`state=JOINED`) and receives a *DIS*, the node will reply with a DODAG Information Object (*DIO*). This allows new nodes to discover existing *DODAGs* in a network, along with obtaining information and configuration parameters of the *DODAG*.

In Figure 2.10 we see that when a node is in the state `JOINING` the transition SendDISReq will be enabled, and the node will be able to send out a DIS request to the network via the place RollToLink. The packet will then be broadcast on the link layer to all the neighbouring nodes. Incoming packets will be on the place Link To Roll. We can see from the SendDIOResponse transition in Figure 2.10 that incoming *DIS* packet will trigger nodes in the state of `JOINED` or `ROOTJOINED` to reply with a *DIO* packet. We are using a transition guard to make the state of the node a requirement, and we use arc inscriptions to tell that the incoming packet has to be of type *DIS*.

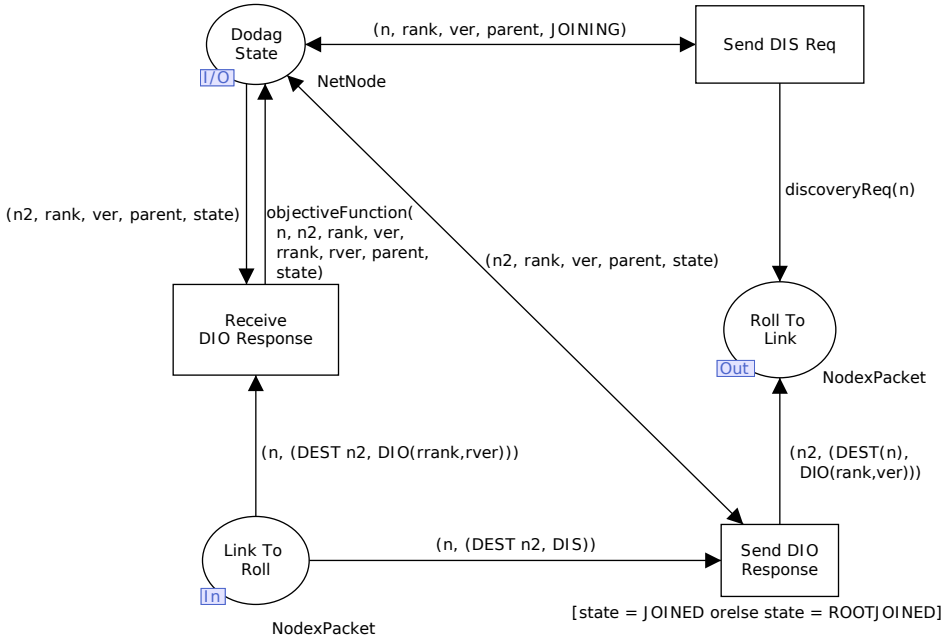


Figure 2.10: The DIS DIO CPN Module

The node receiving the *DIS* packet will respond with a `DIO(rank,ver)` containing the nodes rank and version of a *DODAG*. When the initial node receives a *DIO* response, it will evaluate the response against the information it already has and judge if the incoming *DIO* contains a better suited parent than the current. When we examine Figure 2.10 we see that the state of a node does not matter when receiving a *DIO* response (ReceiveDIOResponse). The reason for this is that *DIO* packets should be evaluated even though a node already has `JOINED` a *DODAG*. This is done so the node can evaluate the incoming *DIO* and check if it contains information that can be used to select a more optimal parent. The optimal parent will be chosen according to an objective function. In our CPN model we have implemented a simplified version of ObjectiveFunctionZero[20] that only takes *rank* and *DODAGVersionNumber* into consideration. The implementation of this function is shown in Figure 2.11.

```
1 fun objectiveFunction(n, n2, rank, ver, rrank, rver, parent, state) =
2   if inconsistency(ver,rver) orelse noDodag(parent,rank,rrank,rver)
3       then 1`(n2, 0, 0, 0, JOINING)
4   else
5     if ((rrank+1 < rank orelse rank=0) andalso not(rrank=0))
6       then 1`(n2,rrank+1,rver,n, JOINED)
7     else 1`(n2,rank,ver, parent, JOINED);
```

Figure 2.11: CPN-ML source code for Objective Function Zero

The objective function zero will check if the current information of the *DODAG* is outdated (i.e: an inconsistency is discovered), check if that the reply contains a valid information about the *DODAG*, and finally check if the information received is better suited as a parent than the current parent is. Based on these criteria, the objective function will either update the current parent or set the node in a *JOINING* state if an inconsistency was discovered.

The DAO module

To discover changes in the network topology and loops in the *DODAG*, Destination Advertisement Object (*DAO*) are used. These are used to transmit and spread information upwards along the directed graph (*DODAG*). This is done by sending a unicast message to the parent of the node. The sender can explicitly require that the receiver should respond with an acknowledgement (*DAO-ACK*) confirming that the *DAO* was received.

Figure 2.12 shows the DAO CPN module. Our CPN model has support for sending both *DAO* requests with and without requests for an acknowledgement. This is done by having a flag in the *DAO* packet, as shown by the arcs going to the socket place Roll To Link. The *DAO* packet has two options: `1` for acknowledgement and `0` for no acknowledgement. The *DAO* module will also process and discard incoming *DAO* requests without the acknowledgement flag set.

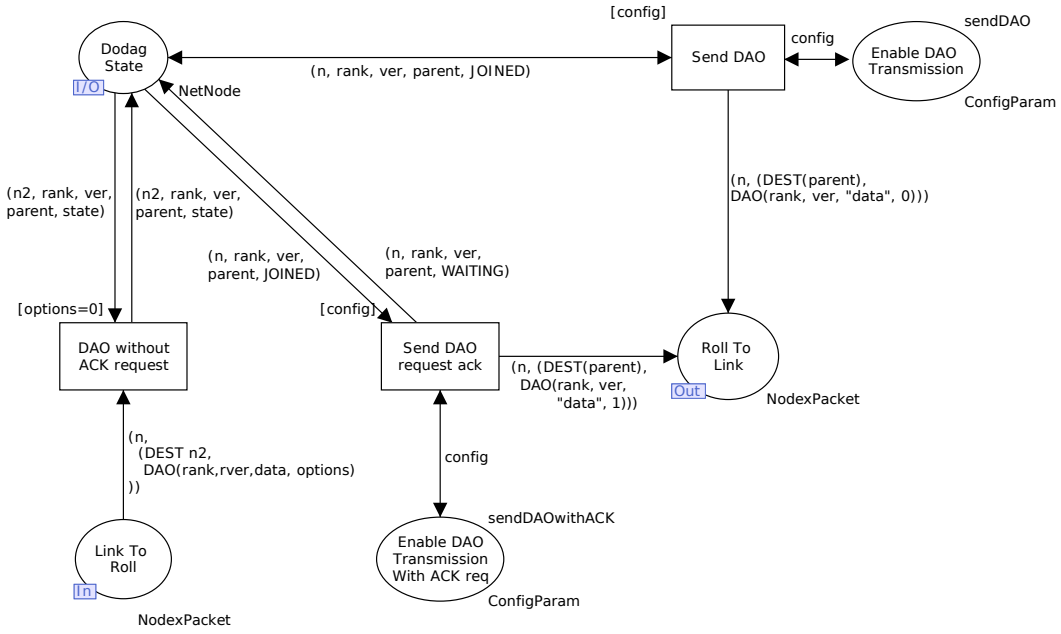


Figure 2.12: The DAO CPN Module

The DAOACK module

Figure 2.13 shows the DAOACK CPN module. When a *DAO-ACK* is not received within a reasonable time-frame, the sender should assume that the link is dead and time out. This will put the node in a state where it wants to rejoin the network (`state=JOINING`). In our model this is done by having a place within the Startup and Timeout module (see Figure 2.14) that will be enabled if the node is in the state `WAITING`. Activating the transition will result in the node going back to the `JOINING` state. When a node receives a *DAO-ACK* packet before timing out, it will evaluate the packet in the same manner as with a *DIO* response, evaluating the response according to the objective function.

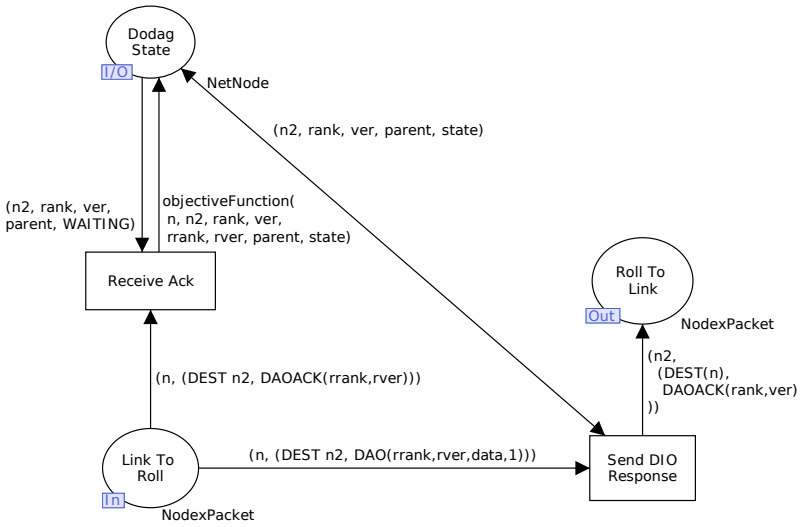


Figure 2.13: The DAOACK CPN Module

The Startup and Timeout module

The node behaviour submodule contains behaviour of the node that is not directly related to sending and receiving packets. The submodule includes the booting sequence of nodes shown as the two transitions Root JOIN and Node JOIN in Figure 2.14. After a regular node boots (`state=INITNODE`), it will try to find a suitable *DODAG* instance to join, and will be in the state `JOINING`. The root nodes will go from a state of `INITROOT` to the state `ROOTJOINED` with some preconfigured values for *DODAGVersionNumber* and *rank*. In Figure 2.14, we have set the rank and version to be initialized to `1` and the parent to `0`, indicating that the root node has no parent. This pre-configuration is shown by the arc inscription `(n, 1, 1, 0, ROOTJOINED)` between the Root JOIN transition and the Dodag State place.

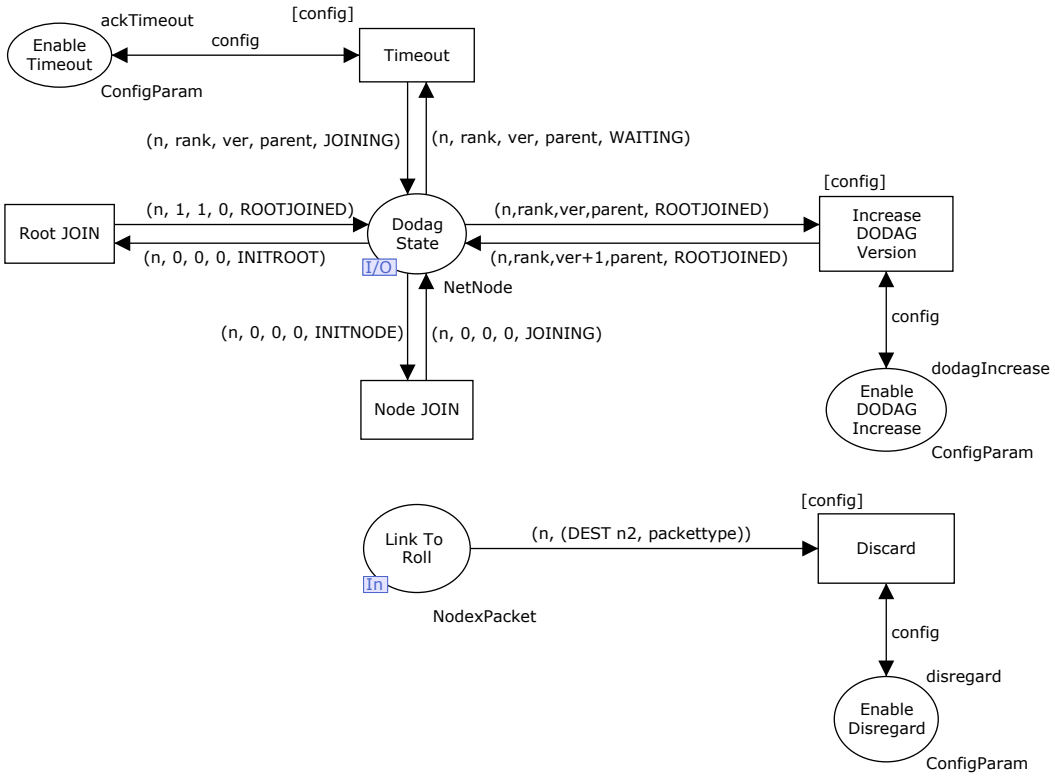


Figure 2.14: The Startup and Timeout CPN Module

The submodule Startup and Timeout has logic that allows the *DODAG* roots to increase the *DODAGVersionNumber*. A version increase will be discovered by nodes sending *DAO* packets with acknowledgement requests (*DAO-ACK*). This will allow the *DODAG* to be rebuilt over time.

2.2.2 Network module

The network representation is divided into two parts. One part that represent the link layer transmission, and one part that simulates changes in the physical topology, creating and deleting links between the nodes. The `TopologyChange` module in our CPN model is largely based on the one used in the DYMO CPN model[14].

Figure 2.15 shows our CPN implementation of the link layer. The place PhysicalTopology contains the nodes and lists of their neighbours in the network. RollToLink is the place of the incoming packets from nodes that should be sent over the network, and LinkToRoll is the destination of the packet. The transition LinkLayer represents the transmission of packets over the link layer.

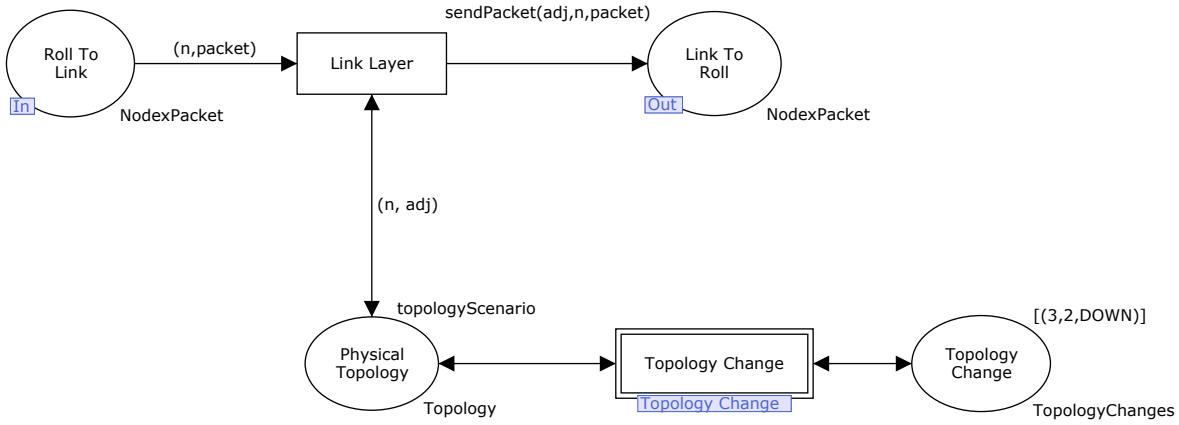


Figure 2.15: CPN-Roll Network Model

Our model allows us to simulate sending packets over the link layer. A packet has a packet type and a destination. The destination can be of two types. It can be a broadcast to all the neighbours in the network, or a unicast to a single node. In our protocol a broadcast is used when a node is probing the network for information about nearby *DODAGs* (broadcasting a *DIS* packet). A unicast is used by nodes which are currently in a *DODAG* and want to send packets to their parenting node (*DAO*) or respond to acknowledgement requests (*DAO-ACK*).

Figure 2.16 shows the different states a node can be in. The node will start in the INIT state and after booting, the node will try to join (JOINING) a *DODAG* by broadcasting DIS packets. When a DIO packet with information about a favourable parent is received, the node will join the *DODAG* and be in the state JOINED. Nodes in the JOINED state can send DAO with acknowledgement requests, putting the node in a WAITING state. The node will then either timeout and try to rejoin (JOINING), or receive an acknowledgement (DAOACK) and go back to the JOINED state.

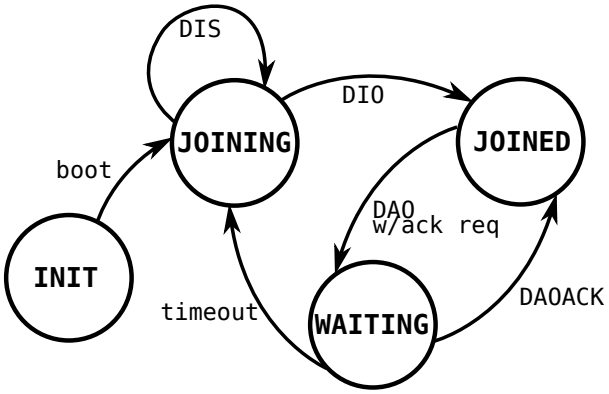


Figure 2.16: State diagram for the Roll Protocol

Chapter 3

The nesC Programming Language and the TinyOS Platform

This chapter introduces the key concepts of the nesC programming language and the TinyOS platform. To introduce nesC and TinyOS we show how selected parts of the Roll Protocol can be implemented in nesC. Our aim is not to give a complete introduction to nesC and TinyOS. The reader is referred to [2] for a complete introduction to these technologies.

TinyOS is a specialized operating system that targets devices with very limited hardware capabilities such as nodes in sensor networks. TinyOS is using a programming language dialect of C named nesC which targets the development of software systems with constraints on processing power and memory usage.

3.1 The nesC Programming Language

To compile software for the TinyOS platform, the nesC compiler is used. A collection of nesC files are compiled by the nesC compiler into a single native C file. This file is then compiled to binary code by a C compiler such as the GNU Compiler Collection (GCC).

3.1.1 Overview

The nesC programming model consists of two main parts. A *configuration* file and nesC *components*. A nesC *component* has information about which *interfaces*

it provides and uses. Interfacing in nesC is a way of structuring the software architecture of an application. The purpose of the configuration file is to *wire* (connect) together components by using these interfaces. There is no static or dynamic linking between files and libraries in nesC. Everything is in the same global namespace and are wired together by the configuration file.

Figure 3.1 shows the relationship between components in a simplified TinyOS implementation of the Roll Protocol. The squares represent nesC components, and the triangles represent interfaces. A triangle inside of a square is an interface that the component is providing, while a triangle outside of a square represent an interface that the component uses. In this model, we have four components. MainC provides the Boot-interface used by the application as an entry point. The RPLProtocolC component constitutes the main program and the DAOC and DIOC components implement the processing of the network packets.

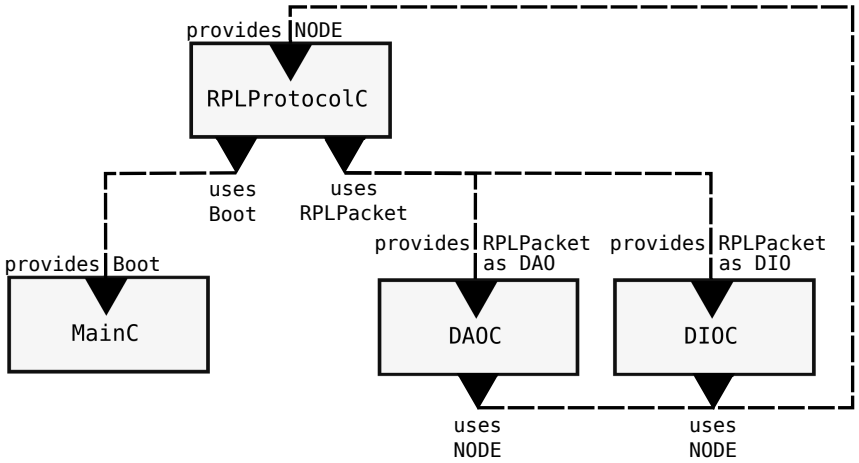


Figure 3.1: Simplified TinyOS implementation of the Roll Protocol

The DAOC and DIOC components are handling incoming packets of the Roll Protocol. To be able to decide on what to do with the incoming packets, the components need to know which state the node is in. In the CPN Roll Protocol model (Figure 2.5) this information is stored in the CPN place DodagState. In our TinyOS Roll Protocol implementation, this information is stored in the RPLProtocolC component. The packet processing components (DIOC / DAOC) use the NODE interface that RPLProtocolC provides to access information about the current state, rank, parent, and *DODAGVersionNumber* of the node.

3.1.2 Type Declarations

To be able to make a TinyOS implementation of the Roll Protocol, we first need to define the basic state type using nesC type definitions.

We can see from Figure 3.2 that nesC type declarations are very similar to that of the C programming language. The *uint* is platform-dependent, as different hardware uses different types of endians. nesC provides a set of platform-independent types. Types prefixed with *nx_* are big-endian values, while types prefixed with *nxle_* are little-endians.

```
typedef enum {
    INITNODE = 0, NODE = 1, JOINING = 2, JOINED = 3, WAITING = 4 ..;
} State;
typedef struct {
    uint8_t id;
    uint8_t rank;
    uint8_t dodagN;
    uint8_t parentId;
    State state;
} NetNode;
```

Figure 3.2: Source code of DataTypes.h

When working with TinyOS applications, memory usage should be taken into account. One way to save memory is by using enums. We should be careful not to use larger integers than we have to, as sensor nodes often have very limited physical memory available.

The header files are very similar to the C header files. The biggest difference is that the nesC header files are global, meaning that if you have included a header in one of your components, it will be included globally for all to use. This is a direct result of how the nesC compiler works: all the nesC components are compiled into a single C source file before being compiled into binary code. Even though this is true, it is a good idea to always include the header file declaration for clarity in the components that are using the custom type definitions.

3.1.3 Program Control-flow

Components communicate with split-phase events as the application is compiled to binary code that typically is used by non-blocking hardware platforms. This means that the signal that initialises an event completes immediately. When the event is done processing the request, it sends a callback to the components implementing the event-handler. This is implemented in nesC using the keywords *signal* and *event*. To invoke a function from another component nesC uses the keyword *call*. To attach an event-listener, the keyword *event* is used, and to trigger an event, the keyword *signal* is used.

The RPLProtocolC component in 3.3 uses two interfaces (as illustrated by Figure 3.1): Boot, which is a standard TinyOS interface that will give a callback when the device has booted, and RPLPacket (Line 5-6) which is a interface for sending and receiving Roll packets. The RPLProtocolC component also provides a simple interface (NODE, Line 3) for accessing the state of the node. The keyword *command* defines a function which can have function parameters and return attributes.

```
1  #include "DataTypes.h"
2  module RPLProtocolC {
3      provides interface NODE;
4      uses interface Boot;
5      uses interface RPLPacket as DAO;
6      uses interface RPLPacket as DIO;
7  }
8  implementation {
9      NetNode node = {
10         .id = 1, .rank = 0, .dodagN = 0, .parentId = 0, INITNODE
11     };
12     command State NODE.getState() { return node.state; }
13     command void NODE.setState(State state) { node.state = state; }
14
15     event void Boot.booted() {
16         // Example scenario
17         Packet packet = { .src = 2, .dest = 1, .packet = DAOpack };
18         call DAO.receive(packet);
19     }
20     event void DAO.send(Packet packet) { ... }
21     event void DIO.send(Packet packet) { ... }
22 }
```

Figure 3.3: Ported source code of RPLProtocolC component

We can see that the RPLProtocolC component listens to three events. The first one is the `booted()` event (Line 15) triggered by the Boot interface. This will be triggered when the TinyOS booting process is done. The next two events are callbacks from the components handling DAO and DIO Roll packets. The `DAO.send(Packet packet)` event (Line 20) will be triggered when the DAOC component signals the `send(packet)` event. Similarly, the `DIO.send(Packet packet)` event (Line 21) will be triggered by the DIOC component.

The RPLProtocolC component does not state which components the DAO and DIO interfaces are connected to. This is done by using the concept of *wiring* and is defined in the application *configuration* file.

3.1.4 Interfaces

An interface in nesC describes the relationship of functions and events between two or more components. The interfaces works quite similar to the interfaces in Java. A nesC interface describes events that could occur (and that should be handled) and commands that are available for use.

To make the intent of an interface clearer it is possible to use the *as* keyword. This keyword is required to be able to distinguish between components that use the same interface. An example of this was shown in the RPLProtocolC component where the `RPLPacket` interface is used twice:

```
uses interface RPLPacket as DAO;
uses interface RPLPacket as DIO;
```

Interfaces are defined in nesC source files. The interfaces describes a set of commands (functions) and events that a component provides. Events are expected callbacks and commands are functions that are available for other components to use. Interfaces can be bidirectional, i.e, both provide commands and describe events at the same time.

The RPLPacket interface (Figure 3.4) defines a function `void receive(Packet packet)` that takes a packet as an argument and returns nothing (`void`), this function will be implemented in the component providing the interface. The `send` event defined in the interface will be signaled from the component providing the RPLPacket interface. Components using the interface will have to implement the event listener for the `void send(packet packet)` event, the component using the interface will have access to use the function `RPLPacket.receive(..)` from the component providing the interface.

```
1 #include "DataTypes.h"
2 interface RPLPacket {
3     command void receive(Packet packet);
4     event void send(Packet packet);
5 }
```

Figure 3.4: Source code of RPLPacket component

The NODE interface shown in Figure 3.5 does not use events. The function `getState()` returns a value (*State*) instead of triggering a callback. This can be done safely for functionality that do not require complex or time-consuming operations.

```
#include "DataTypes.h"
interface NODE {
    command State getState();
    command void setState(State state);
}
```

Figure 3.5: Source code of NODE interface

3.1.5 Wiring and Configurations

Because of the nature of the devices running TinyOS it has to be easy to change the implementation of a component. This is done by wiring together different components. The wiring is done in the configuration file of the application. Wiring is a concept used by nesC to connect components. Using a configuration file, we are able to connect the different components by communicating through the interfaces that are used and provided. Since interfaces can be bidirectional both the components wired together are affected by the wiring.

The implementation from the configuration file **RPLProtocolAppC.nc** in Figure 3.6 shows how the components of the application (Figure 3.1) is wired together. All the components of the application are defined by the *components* keyword. In this application we have four components: MainC, RPLProtocolC, DAOC and DIOC.

```
1  configuration RPLProtocolAppC { }
2  implementation {
3      components MainC, RPLProtocolC, DAOC, DIOC;
4
5      RPLProtocolC.Boot -> MainC.Boot;
6      RPLProtocolC.DAO -> DAOC.RPLPacket;
7      RPLProtocolC.DIO -> DIOC.RPLPacket;
8
9      DAOC.NODE -> RPLProtocolC.NODE;
10 }
```

Figure 3.6: Source code of RPLProtocolAppC.nc

The RPLProtocolC is using the Boot interface from the MainC component providing callbacks when the booting process is completed (Line 5). The RPLProtocolC component is also using the two components for processing RPL packets: the DAOC and DIOC components (Line 6-7). DAOC uses the NODE-interface provided by the RPLProtocolC component (Line 9). We see from the configuration file that both DAOC and DIOC provide the same interface but are mapped to two different

name spaces in the RPLProtocolC component.

The line `RPLProtocolC.DAO -> DAOC.RPLPacket` can be read as PRLProtocolC uses the RPLPacket interface found in the DAOC component with the local identifying name DAO. Alternatively it can be read as DAOC provides the interface RPLPacket which is used by RPLProtocolC locally with the name DAO.

To wire together components, three symbols can be used `->`, `<-` and `=`. The `=` wiring symbol is used to rename another components implementation as its own, and hence how the configuration is implemented. The `->` and `<-` symbols are used to wire together already existing implementations. `->` and `<-` syntactical sugar for the same thing, so that:

```
DOAC.NODE -> RPLProtocolC.NODE = RPLProtocolC.NODE <- DAOC.NODE
```

3.2 The TinyOS Platform

TinyOS is used by multiple hardware platforms. The TinyOS platform supports a wide range of *wireless sensor network* (WSN) platforms, microprocessors and peripherals[5]. The *TinyOS Enhancement Proposals*[4] (TEPs) are a set of documents discussing modifications and enhancements of the TinyOS platform. The TEPs deal with best practices, hardware abstractions, and TinyOS libraries and programming interfaces.

3.2.1 APIs

The TinyOS platform provides a wide range of pre-defined components that are generic and platform-independent, making them applicable for a wide range of hardware platforms. These components include interfaces and abstractions for data structures, communication, working with timers, storing information, logic for booting, and utilities such as random number generators. For a complete list of interfaces and components see the TinyOS homepage[2].

Data Structures

TinyOS has a set of predefined platform-independent data structures, a few of them being:

- First in first out queues (*FIFO*) with bounded sizes such as the `QueueC.nc` and `BigQueueC.nc`.

- Bit arrays for compactly storing bits that support atomic operations (`BitVectorC.nc`).
- A dynamic memory pool (`PoolC.nc`) that when initialized have a maximum number of items (size). New items can be added and old items can be removed, but the pool can never grow larger than the initial given size.
- State machines for sharing states (`StateC.nc`) between components. This allows components to track the state of other components through a common interface, and can be used for keeping track of multiple states at the same time. These state machines have a `S_IDLE` state, and two methods of changing the state. The `requireState(...)` method can require a state change and will only succeed if the state machine is idle (`state = S_IDLE`). The second method, `forceState(...)`, will change the state of the state machine regardless of the previous state.
- The TinyOS `message_t` message buffer. A message buffer for sharing data over link layers that is optimized for compact data storage. The `message_t` consists of a header, some data, a footer, and meta-data. The header, foot and meta data all must be external structs (`nx_struct`) to ensure cross-platform compatibility.

Communication

There already exist many implementations of network protocols for the TinyOS platform and the *TinyOS 2.0 Network Protocol Working Group*[3] (net2) is responsible for defining programming interfaces for network protocols. There are also interfaces for using serial ports communication and radio links.

The radio communication interfaces allow us to send and receive radio packets. The Active Message interfaces provide abstractions for sending (`AMSendC.nc`) and receiving (`AMReceiveC.nc`) radio messages by using the `message_t` message buffers.

The net2 working group has specified and implemented a number of protocols. They have created an IPv6 implementation for the TinyOS platform, the *Berkeley Low-powered IP stack* (BLIP), which other protocols build upon. The most notable protocols being the Roll Protocol (Chapter 2.1) and the *Constrained Application Protocol* (CoAP). CoAP is a protocol used for application to communicate and transfer data over the web by using a subset of the REST software architecture.

Other available network protocols include the *Collection Tree Protocol* (CTP) used to provide a many-to-one network layer by delivering packages to a sink in the

network, and the *Link Estimation Exchange Protocol* (LEER) for discovering and exchanging information about the network quality between sensor nodes.

Serial communication is used for communication between sensor nodes and computers. By using serial ports, it is possible to interact with the nodes and read sensor data as well as inspect network packets. The sensor nodes can use the computers as proxies to communicate with the outside world.

Utility

TinyOS comes with a set of platform independent utilities. There are components that allows us to mount (`MountC.nc`) physical volumes such as SD-cards and hard drives, and store information locally. Components for handling time and timers that allow support for preforming actions at given intervals. Components for interacting with hardware, such as the `LedC.nc` which is used to interact with led-lights connected to the hardware devices. There are interfaces that make it possible to read data (`ReadC.nc`) from attached devices such as light sensors, temperature sensors, and accelerometers. Other utilities include random number generators (`RandomC.nc`), caching mechanisms (`CacheC.nc`), and resource sharing (`SimpleArbiterC.nc`).

3.2.2 TOSThreads

The *TOSThreads threading library* allows easy use of threading on the TinyOS platform with the efficiency of the event-based software architecture that the TinyOS platform provides. TOSThreads allows for synchronous and asynchronous code to be executed. The library does not break with the non-blocking and event-based execution model of the TinyOS platform. The scheduler allows for asynchronous code to call synchronized code but not the other way around.

To be able to use the TOSThread library, we must change the booting sequence so that TOSThreads is the thread scheduler that takes control of the microcontroller. The application has to include a `chip_thread.h` for the given architecture we are implementing the components for.

The *TOSThreads thread library* consist of five main parts: The *TinyOS task scheduler* which prioritizes and allocates resources to the threads. A *single kernel-level TinyOS thread* that has the highest threading priority. This is to preserve the timing-sensitive operations of the TinyOS platform. This thread will get priority as long as the TinyOS task queue is non-empty. A set of *user-level application threads* that the TinyOS task scheduler will execute until the TinyOS task queue is no longer is empty or all of the user-level threads are either done executing, waiting on

synchronizing or blocking on I/O operations. The TOSThreads has a well defined *Application Programming Interface* (API) with methods for creating, pausing, resuming and destroying threads. The TOSThreads comes with a corresponding *implementation* of this API.

3.2.3 TOSSIM

TOSSIM is a sensor node simulator for TinyOS. Using TOSSIM we are able to simulate how nesC code behaves without having to install the code onto a physical device. TOSSIM is compiled to a shared library that comes with a set of standard functions for simulating behaviour. The TOSSIM simulator has full support for the *Micaz* platform and some support for the *Mica2* platform.

To be able to run the simulation, we have to compile the nesC application with the `sim` flag. This will generate a shared library file (`_TOSSIMMoudle.so`) and a python interface (`TOSSIM.py`) for the library. The generated library allows us to set up an instance of TOSSIM where we can create new sensor node and simulate a real environment. The framework gives us access to simulate the network at the bit level along with packet loss and corrupt packets, and precise measurement of time and interrupts.

Figure 3.7 shows how we use the `dbg(CHAN, msg)` method to create debug channels that allow us to log events and behaviour. In Line 3, we add a debug channel called `state` that will be invoked when the nesC method `DODAG.setState(...)` is called during simulation.

```
1   NetNode node;
2   event void DODAG.setState(State state) {
3       dbg("state", "%s RPL | State change: %i -> %i.\n",
4           sim_time_string(), node.state, state);
5   }
6
7   event void Booted.booted() {
8       dbg("boot", "%s RPL | Application booted.\n",
9           sim_time_string());
10          DODAG.setState(JOINING);
11  }
```

Figure 3.7: Snippet of the RPLProtocolC.nc component showing debug messages

The python script (`simulation.py`, shown in Figure 3.8) imports the TOSSIM library (Line 1), and opens a new file used for logging the events (Line 2). We

```
from TOSSIM import *
log = open("log.txt", "w")

sim = Tossim([])                # creating simulation environment
sim.setTime(0)
sim.addChannel("boot", log)     # listening to boot dbg msgs
sim.addChannel("state", log)    # listening to state dbg msgs

m0 = sim.getNode(0)             # creating a node
m0.bootAtTime(300)              # booting node
for _ in range(5):
    sim.runNextEvent()          # running simulation step
```

Figure 3.8: Source code of the python simulation script

set the simulation time to start at `0`, and then bind the logfile to the two output channels `boot` and `state`.

Given a scenario where the nesC program boots and changes state from `0` (`INITNODE`) to `2` (`JOINING`) the log file (log.txt) would contain the following two lines:

```
DEBUG (0):  0:0:0.0000000300 RPL | Application booted.
DEBUG (0):  0:0:0.0000000300 RPL | State change:  0 -> 2.
```

The `DEBUG (0)` indicates that the *id* of the node invoking the `dbg()` method is `0`. The information following is the time of invocation (`0000000300`) and the debug message (`RPL | Application booted`).

The TOSSIM simulator allows us to create nodes (`getNode`), running simulation events (`runNextEvent`), accessing the media access layer (`MAC layer`), accessing the network and radio (`radio`), and adding channels for output (`addChannel`). For a complete list of functionality see the TOSSIM API[2].

Chapter 4

CPN Model Refinements

In this chapter we describe how the CPN Roll Protocol model has been refined. The refinement has been done by adding TinyOS platform specific details to the CPN model, and the reader will be given an overview and motivation for each of the refinement steps.

4.1 Model Refinement Overview

To be able to generate code from an abstract platform independent model, we have to refine the model so we can make accurate assumptions about how to translate the model into code specific to a given platform. We can do this by either making details in the model more explicit, or by introducing conventions for how the model should be structured.

We started with creating a CPN model representing the behaviour of the Roll Protocol. The original CPN model used in the case study was created prior to having domain knowledge about TinyOS as a target platform. The motivation for this was to keep the CPN model as platform independent as possible. Starting with an abstract model created without prior domain knowledge would give us a stronger indication that the refinement approach is generic and can be applied for other similar problems. The CPN model was created with a hierarchy of three levels (see Chapter 2.2). The main level contains two parts: one for the network, and one for the Roll Protocol.

The refinement approach that we have developed consist of five steps that can be used to add details to network protocol models created with CPNs. These steps allow us to use the resulting refined CPN model to generate the structure and the behaviour of the corresponding TinyOS network protocol application. The generated behaviour does not include the translation of CPN ML expressions

to nesC code. Tools for automatically translating Standard ML to C already exist and have been implemented[19]. For the refinement steps we have chosen to structure the protocol module (RollProtocol, Figure 2.3) into three levels. At the top level there is an application containing components. These components are what constitute the application, and each of these components have an internal structure of events and commands. These do in turn have an internal structure describing the behaviour of how they are executed on the target platform. The five refinement steps are:

Step 1: Component Architecture
The first step is to define the architecture of the application by annotating CPN submodules corresponding to TinyOS components, and by inscribing the connected CPN arcs with text specifying which TinyOS interfaces they are using and providing.

Step 2: Resolving Interface Conflicts
The second step is to resolve interface conflicts. This is done by introducing the `as` keyword allowing components to use multiple instances of a single interface by assigning unique local names.

Step 3: Component and Interface Signature
The third step is to add type signatures to components and interfaces. We do this by separating the CPN logic into submodules representing TinyOS events and commands. We generate a TinyOS header file that maps CPN colour sets to nesC types, and we use the colour set of the connected CPN places to describe the interface signatures.

Step 4: Component Classification
The fourth step is classifying component types. After defining the interface signatures we have a specification of the event and command invocations. We classify the application components into four main types: timed components, external components, application boot components, and general components.

Step 5: Internal Behaviour
In the fifth and final step of the refinement, we define the internal behaviour of nesC commands and event handlers. We model a control flow in the CPN model that allows us to generate the behaviour of TinyOS commands and events.

We use pragmatics as syntactical annotations for the CPN model to direct the code generation. The pragmatics add explicit information to the CPN model, and are used to describe details about the target platform. The extra information makes the CPN model more expressive in terms of linking elements and structure to the target platform, and reduces the number of assumptions needed to be taken during the code generation (Chapter 5). The pragmatics are text strings that can be used to annotate arcs, transitions, substitution transitions and places.

We have chosen to use `<<type (param1, param2)>>` as the format for describing pragmatics, which is the same format as the one described in [18]. The `type` describes the pragmatics type, and the `(param...)` describes extra information about the given pragmatic. In our final refined model, we have pragmatics for describing `components`, `events`, `commands`, `tasks`, and a set of pragmatics for describing the internal behaviour of events and commands.

4.2 Step 1: Component Architecture

The first step of the refinement process is to find a way to structure the CPN model so that we can relate CPN modules to TinyOS components. Figure 4.1 shows the original CPN Roll Protocol model (see Chapter 2.2). The original model is split into four submodules and three places. The submodules (DISDIO, DAO, DAOACK, and StartupandTimeout) contain the logic of the Roll Protocol. The LinkToRoll socket-place is used for receiving network packets, RollToLink is used for sending packets, and the local place DodagState contains the current information about the network that a node implementing the Roll Protocol has available.

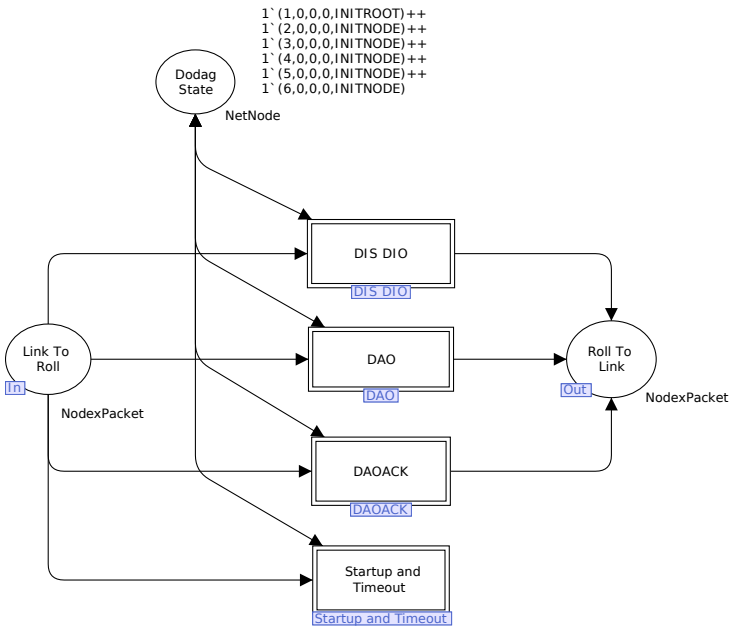


Figure 4.1: Roll Protocol Module - Original

We discovered that both CPN submodules and TinyOS components were encapsulating logic, and for code generation it would be reasonable to assume that TinyOS components could be represented as CPN submodules. TinyOS applications are

built with components, and these components are wired together by an application configuration file to describe connectivity (see Chapter 3.1.5). CPNs submodules exchange tokens (data) between places, and from this we could draw a connection between TinyOS interfaces and CPN socket-places.

The hierarchical structure of CPN models and submodules can be refined to resemble the component layout of TinyOS applications. The purpose of the first refinement step is to specify what CPN submodule is to represent which TinyOS component, and how the components are connected to each other. The `<<component>>` pragmatic is introduced to describe the relationship between a TinyOS component and a CPN module. We annotate the CPN arcs with text to describe what interfaces each of the components are using and providing. This is done by inscribing arcs connected to the submodules with interface names. An arc going into a CPN submodule is indicating that the TinyOS component is providing that interface, while an outgoing arc is indicating usage of the interface.

Figure 4.2 shows the result of the first refinement step which affects only the top-level of the CPN model. The substitution transitions are annotated with the `<<component>>` pragmatic, and the arcs connected to the substitution transitions are annotated with which interface they are connected to. The arc going from LinkToRoll to DISDIO in Figure 4.2 is inscribed with the text NetPacketInterface. This is a representation of the interface that would be provided by the generated DISDIO TinyOS component. The StartupAndTimeout submodule is discarding packets received after timing out, and does not process packets the same way as the other submodules, which is why it does not have an associated interface specification.

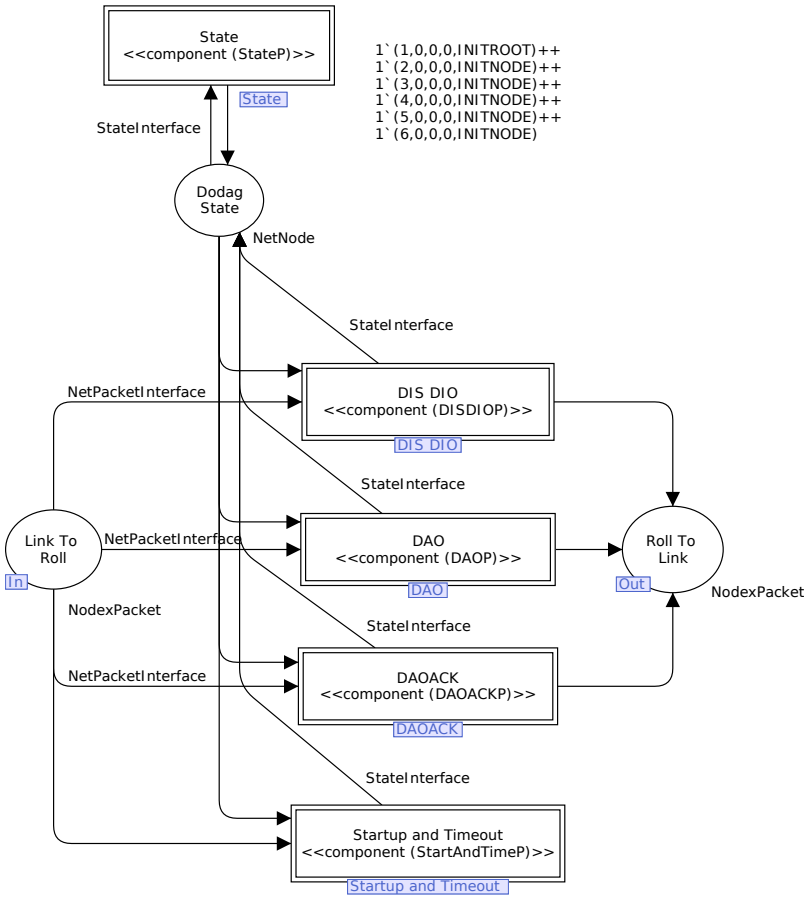


Figure 4.2: Roll Protocol annotated with TinyOS interfaces and components

Figure 4.3 shows a snippet of the code generated based on the pragmatics in Figure 4.2. Line 3 lists the CPN submodules annotated with the `<<component>>` pragmatic and line 5-8 shows a sample of the application wiring. The RPLProtocolP TinyOS component is representing the CPN module containing the components, while Line 11-12 shows the generated interfaces.

The first CPN model refinement allows us to generate a simple structure of the TinyOS application with information about the components, and how they interact through interfaces. The generated code is still missing a lot of required details to be operational. As an example, if the same interface is used multiple times by a single component there is no way to differentiate between which interface is provided by what component (e.g. DISDIO and DAO both provide the `NetPacketInterface`). The inscription on arcs can also be hard to understand as there is no explicit annotation as to whether the interface is used or provided. Furthermore, the generated code contains no types, commands or events.

```
configuration RollProtocolAppC { }
implementation {
    components DISDIOP, DAOP, DAOACKP, StartAndTimeP, StateP,
       RollProtocolC;

    DAOP.DodagState <- RPLProtocolP.DodagState;
    DAOP.LinkToRoll <- RPLProtocolP.LinkToRoll;
    RPLProtocolP.DodagState -> DAO.DodagState;
    RPLProtocolP.DodagState -> DAOACK.DodagState;
    ...
}
interface NetPacketInterface { }
interface StateInterface { }
```

Figure 4.3: Generated TinyOS Configuration Code

4.3 Step 2: Resolving Interface Conflicts

The second step of the refinement is to resolve ambiguities in the way interfaces are described. The CPN socket place LinkToRoll, (see Figure 2.3 and Figure 2.5) connecting the top level of the CPN model to the protocol submodule is acting as both an external interface for the CPN network module, and as a local interface for the submodules in the protocol module. The second issue with interfaces after the first refinement step is that a single component cannot use multiple instances of a single interface provided by different components.

To resolve the ambiguities where socket places was used as multiple interfaces, we add an additional transition between the incoming socket place and the places going into submodules. Figure 4.4 shows the CPN Roll Protocol after the second step of the refinement. We have split the LinkToRoll into two places, this is to differentiate between the externally received packets (tokens added to the LinkToRoll place), and the packets that are processed within the application. The second step in resolving interface ambiguity is to allow TinyOS components to use multiple instances of a single interface. This is resolved by adding an `as` keyword to the arcs that allows us to give interfaces local unique names. From Figure 4.4 we see that some of the interface declarations on incoming arcs of substitution transitions contain the `as` keyword. This allows a single component to use multiple instances of the same interface. We have also introduced some syntactical sugar for the annotations allowing us to use the keywords `provides` and `uses` for readability.

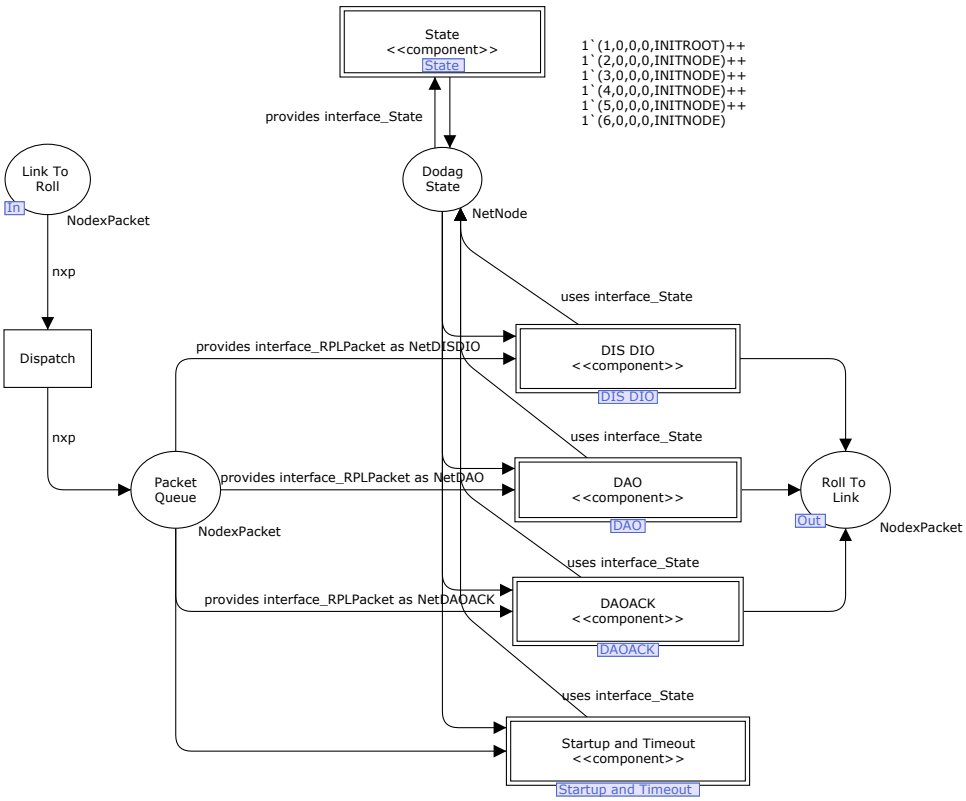


Figure 4.4: Roll Protocol ambiguities resolved

Figure 4.5 shows the generated nesC code after the second refinement step. The TinyOS wiring reflects the use of interface aliases, and the wiring in Line 4-6 connects the interfaces provided by DISDIO, DAO and DAOACK to the RollProtocol component. Line 9-12 describes how the RollProtocol component uses a single interface with three distinct local names, i.e, the interface `interface_RPLPacket` is given a unique alias for each instance. The wiring specifies how the aliases are mapped to other components.

The second refinement of the CPN model has resolved the ambiguities with providing and using multiple interfaces, as it has made the interface annotations more readable. The generated code contains the application structure and reflects how the components uses and provides interfaces. The generated nesC code does not provide details on what command and event signatures the interfaces include.

```
configuration RollprotocolAppC { }
implementation {
    ...
    RollProtocol.NetDISDIO -> DISDIO.interface_RPLPacket;
    RollProtocol.NetDAO -> DAO.interface_RPLPacket;
    RollProtocol.NetDAOACK -> DAOACK.interface_RPLPacket;
} ...
module RollProtocolC {
    provides interface interface_RPLPacket as NetDISDIO;
    provides interface interface_RPLPacket as NetDAO;
    provides interface interface_RPLPacket as NetDAOACK;
} ...
```

Figure 4.5: Generated component with multiple usage of a single interface

4.4 Step 3: Component and Interface Signatures

To be able to generate a meaningful application structure for the TinyOS platform from CPNs, we need a way to describe methods and types used in the interfaces. The third refinement step introduces types, events and commands to the CPN model. Here we exploit a relationship between CPN places and TinyOS interfaces, CPN places are used for moving tokens between CPN submodules and can be viewed as a representation of TinyOS interfaces. Each of the CPN places connected to the substitution transitions annotated with the `<<component>>` pragmatic has a type (colour set) that can be translated into a nesC type.

We introduce the `<<interface>>` pragmatic which we use to make explicit the relationship between CPN places and TinyOS interfaces. By introducing this pragmatic, we no longer have to make the assumption that all connected places are interfaces. We use the colour set of a CPN place to describe the interface signature. When looking closer at the relationship between CPN colour sets and nesC types, it can be seen that colour sets can be translated into nesC types by making assumptions about how primitives are translated. The translation of types is described in detail in Chapter 5.2.1.

Figure 4.6 shows the generated nesC structs and the corresponding colour set declarations for the `NodexPacket` and the `Packet` types. We were able to translate colour sets defined in the CPN model automatically to their respective nesC types.

The original CPN model was modelled in a way that made it hard to distinguish between the behaviour of events and commands. This is because there was a small number of transitions and places modelling much of the behaviour, i.e, it was modelled in a very compact manner. A transition could execute multiple

```
colset Packet = product          |      typedef struct {
     Dest * PacketType;          |          Dest dest;
                                 |          PacketType packettype;
                                 |      } Packet;
colset NodexPacket = product     |      typedef struct {
     Dest * Packet;              |          Nodes nodes;
                                 |          Packet packet;
                                 |      } NodexPacket;
```

Figure 4.6: Sample of generated nesC header file and corresponding colour sets

actions at once via complex expressions on the outgoing arcs, and those actions could in turn manipulate the tokens at multiple places.

Figure 4.7 shows the original DISDIO module which contains logic for handling both DIS and DIO packets. The CPN submodule can receive packets from the LinkToRoll place and based on the packet-type, it will either send a response by enabling the SendDIOResponse transition or update the node status by enabling the ReceiveDIOResponse transition, and the module could send DIS (the SendDISReq transition) requests regardless of it receiving packets or not. The CPN module shown in Figure 4.7 illustrates how it can be difficult to know what behaviour corresponds to which TinyOS commands or events, as there is no explicit information in the module specifying this. The solution to this problem is to encapsulate the logic for events and commands in their own submodules as shown in Figure 4.8.

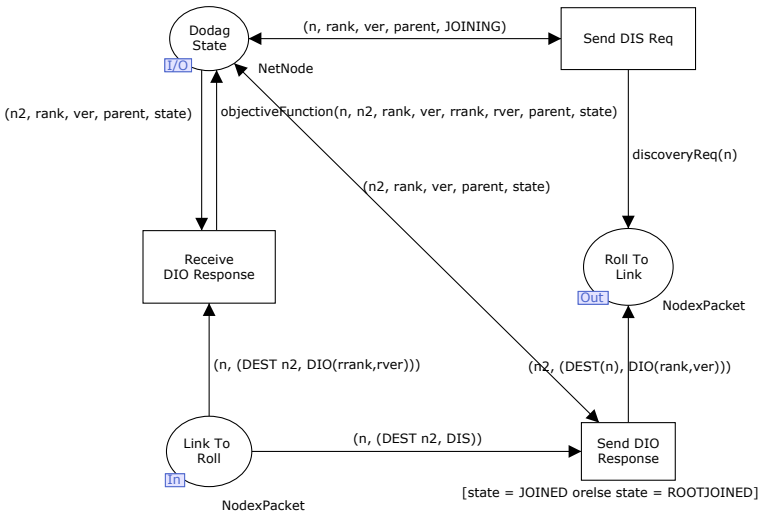


Figure 4.7: DISDIO Module - Original

Figure 4.8 shows the model after the third refinement step. Each event is rep-

resented as a substitution transition annotated with the <<event>> pragmatic. The submodules contains the logic associated with the corresponding event. The arcs describe the relationship to other interfaces, and the places connected to the submodule describes interface method signatures. Arcs going into a submodule with the <<event>> pragmatic correspond to event invocations, outgoing arcs are interface calls with no return value, and bidirectional arcs are invocations of a command or an event that returns a value.

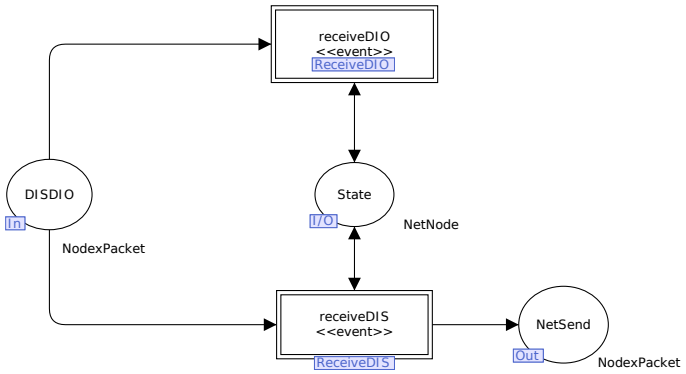


Figure 4.8: DISDIO Module - refined structure

At this point, we are able to generate the structure of the interfaces and the header file needed to describe the TinyOS types. We can also generate the interaction between the components. This is done by assuming that events have no return value, and that all places connected to submodules annotated with the <<event>> pragmatic are interfaces.

Figure 4.9 shows the generated code for the interface provided by the DISDIO component. The interface contains all the events and commands in a submodule annotated with the <<event>> or <<command>> pragmatics. Line 1 includes the generated header file containing the nesC types translated from the colour set of the CPN model. Line 3 corresponds to the arc between DISDIO and receiveDIO in Figure 4.8, and Line 4 to the arc between DISDIO and receiveDIS.

```
1 #include "global.h"
2 interface DISDIO {
3     event void receiveDIO(NodexPacket var_nodexpacket);
4     event void receiveDIS(NodexPacket var_nodexpacket);
5 }
```

Figure 4.9: Generated nesC Interfaces with Types

During the model refinement of events and commands, we identify the functionality of the network protocol that is not triggered by external events (e.g. by receiving

a network packet). At this point in the refinement we are not able to differentiate between externally triggered events and events that should be executed at regular intervals.

4.5 Step 4: Component Classification

Not all events are invoked by external calls, and the fourth refinement step is to classify component into component types. The observation that some of the functionality would be invoked by external calls, run a single time at application boot, and that some functionality would be triggered at regular intervals motivated us to classify components into five distinct types. The five classifications are: events that should be triggered at startup `(boot)`, tasks that should be run at a given interval `(timed)`, externally provided components `(external)`, a dispatcher `(dispatch)` that would parse network packets, and regular components triggered by event or command invocation.

We added a `timed` parameter to the `<<component>>` pragmatic, which is used to annotate components containing tasks that should be triggered periodically. We added a component for timed tasks, and moved the events and commands that would not be triggered externally into this component. The submodules that are to be triggered regularly are annotated with the `<<task>>` pragmatic. The timed component in the refined CPN model (Figure 4.10) has functionality for sending DIS and DAO packets, for increasing the DODAG version number, and logic for timing out while waiting for DAO acknowledgements. Figure 4.11 shows the code generated for the interface of the timed task component. Line 6-9 describe the signatures of the TinyOS tasks that the TimedTasks implements and Line 2-3 describes the interface usage of the component, which is reflected in Figure 4.14 by the annotated arcs going into the TimedTasks substitution transition.

Submodules annotated with the `<<component (boot)>>` pragmatic contain logic that will only be run at application boot. In the refined CPN model, the component annotated with the `boot` parameter (Startup, Figure 4.14) contains logic for deciding if the node should boot as a root-node or as a regular node, as shown by Figure 4.12.

The components annotated with `<<component (external)>>` are components that will not be generated by the code generation, and will have to be implemented manually (or connected to an existing TinyOS library component provided by the platform). The components using interfaces annotated with the `external` parameter will be generated with the correct usage of the interface, and the application configuration file will also reflect the fact that components are using the external components.

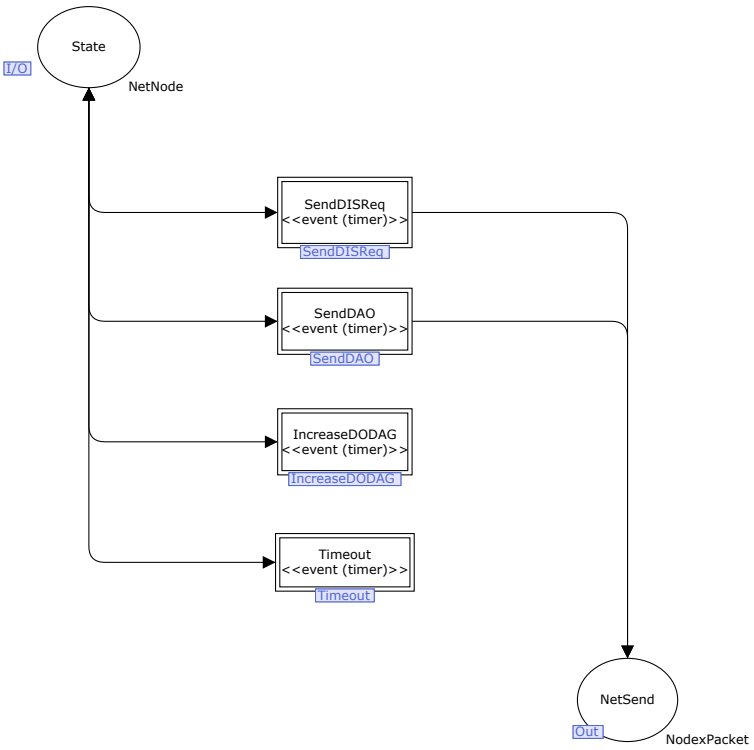


Figure 4.10: Roll Protocol Module - Timed tasks component refinement

```
module TimedTasksC {
    provides interface NetSend;
    uses interface State;
}
implementation {
    task void SendDISReq() { }
    task void SendDAO() { }
    task void IncreaseDODAG() { }
    task void TimeOut() { }
}
```

Figure 4.11: Generated TimedTasks component

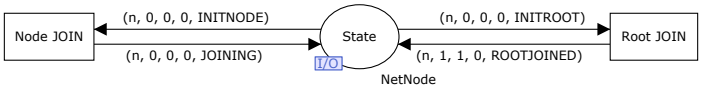


Figure 4.12: Roll Protocol Module - Startup component refinement

The `dispatch` component, as shown by Figure 4.13, is assigned to be a dispatcher for network packets. The dispatcher is an interface towards the external network,

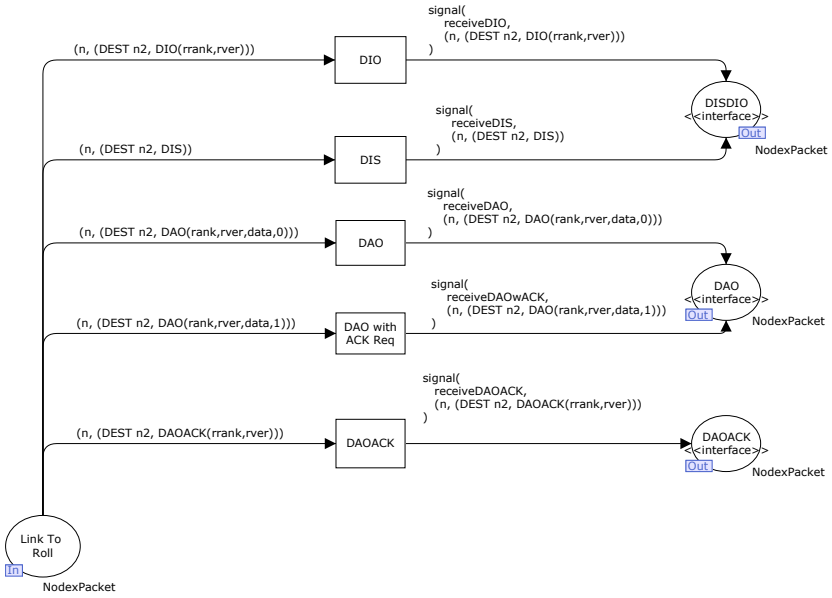


Figure 4.13: Roll Protocol Module - Dispatch component refinement

and receives the network packets and signals the correct component based on which type of packet received. The dispatcher component will signal the packet processing components, and will replace the overlying RollProtocol component from the previous refinement steps. The refined CPN model has the following processing components: the DISDIO component processing DIS and DIO packets, the DAO component processing DAO packets, and the DAOACK processing acknowledgements to DAO packets.

Figure 4.14 shows the refined structure of the CPN Roll Protocol. We have gone from a very compact model to a model with explicit details for the target platform. The structure contains pragmatics to describe different types of components, and CPN places annotated with the `<<interface>>` pragmatic to describe TinyOS interfaces. The arcs between the places and submodules describe the connection between the interfaces and components.

The code generated from interfaces will at this point in the refinement include events and commands with types, the relationship between components and interfaces, and the TinyOS wiring will be generated based on the arcs representing components using and providing interfaces. The first four refinement steps enabled us to use the refined CPN model to generate the structural outline of the corresponding TinyOS application.

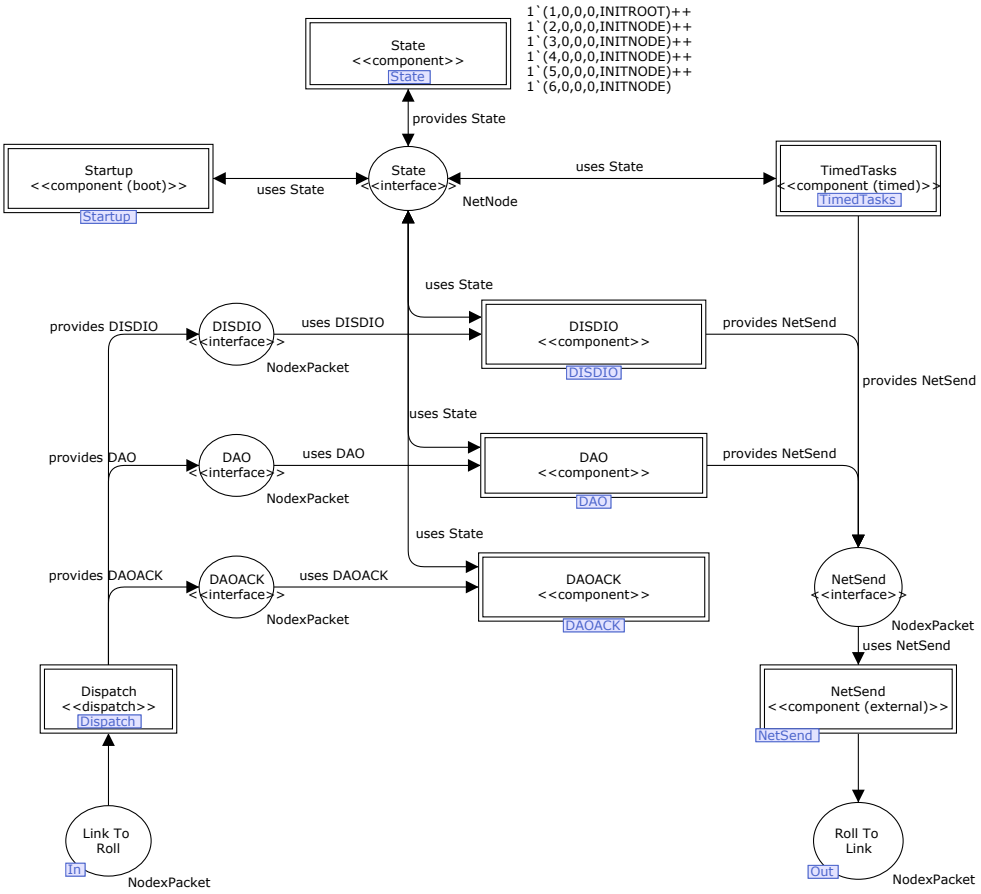


Figure 4.14: Roll Protocol Module - Component Refinement

4.6 Step 5: Internal Behaviour

The final step of the refinement process is to describe how the protocol behaves internally. We introduce pragmatics to describe the control flow where the order of execution is clearly identifiable. By having a token move in a single path between places, we are able to obtain a clearly identifiable control flow that can be exploited for code generation purposes.

Figure 4.15 shows the refined receiveDIO event. The Receive DIO event module is a refinement of the ReceiveDIOResponse transition in Figure 4.7. The transition is connected to two places, namely LinkToRoll and DodagState. We have refined the ReceiveDIOResponse transition by introducing new pragmatics to unambiguously specify the corresponding TinyOS nesC code. We use the `<<ID>>` pragmatic to describe the control flow. By using the `<<ID>>` pragmatic, we can create a clearly identifiable path of execution. The initial place of the control flow, Idle, is identifiable by having an initial marking (`true, bool`). The token is moved down

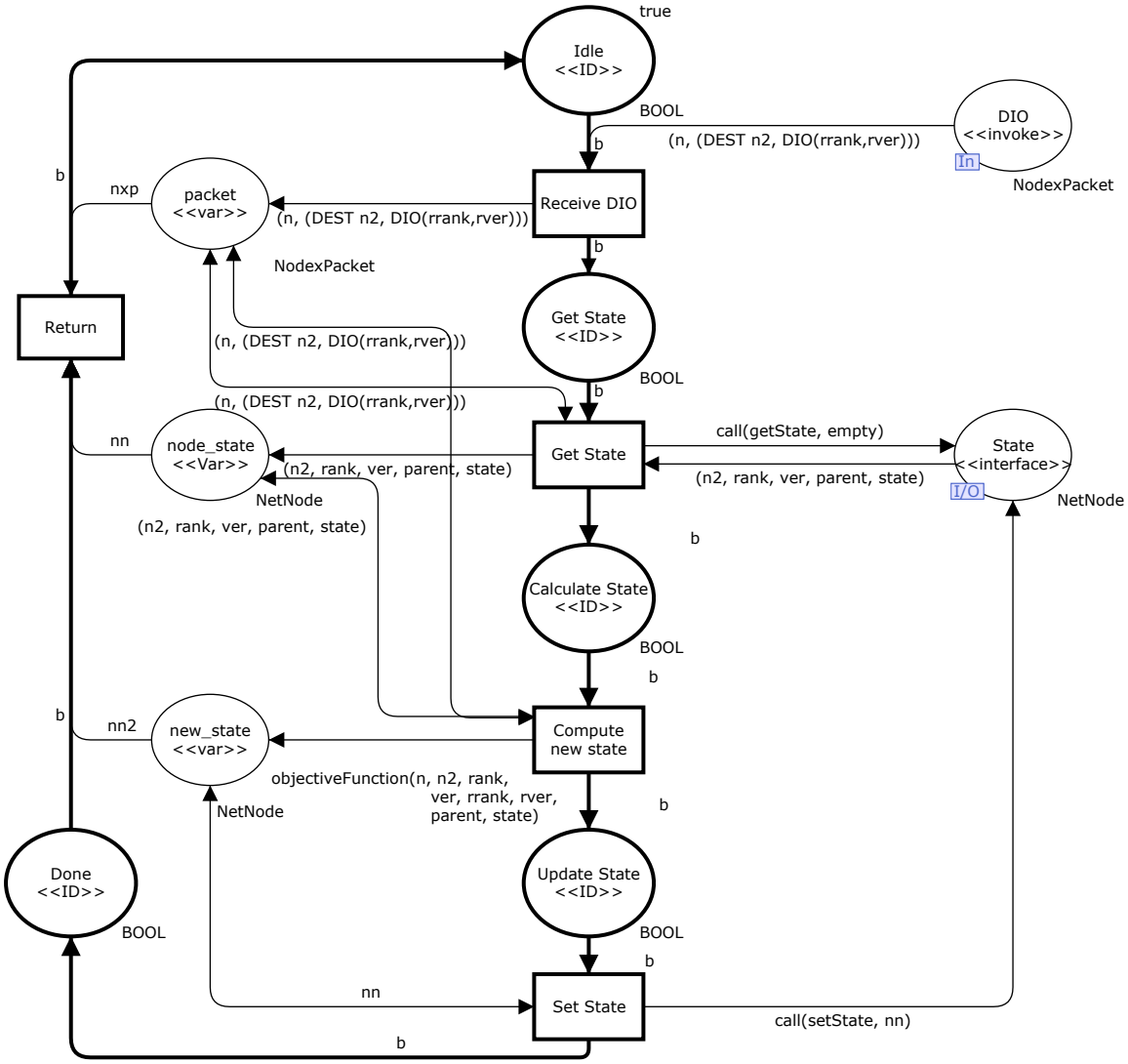


Figure 4.15: Refined CPN model of the Receive DIO event

through the line of places annotated with the `<<ID>>` pragmatic. In each step of the model execution, we use predefined patterns to recognize behaviour. We use the `<<invoke>>` pragmatic to describe the signature of the event or command, and the `<<var>>` pragmatic to store values between the transitions in the control flow. By using a clearly identifiable control flow and pragmatics to add additional information, we can generate the equivalent nesC behaviour. The generation of behaviour is described in detail in Chapter 5.3.

Figure 4.16 shows the generated nesC code of the receiveDIS event (Figure 4.15). By following the control flow and using pattern recognition, we are able to generate nesC statements for each of the transitions in the CPN model. Line 1-3 shows

the outline of the CPN ML `objectiveFunction` used on the arc from transition ComputeNewState to the place new_state in Figure 4.15. The code generator is not able to translate CPN ML to nesC, but it can generate an outline of the function. Line 4 is the generated event signature, and Line 5-7 shows the CPN places annotated with the <<var>> pragmatic (packet, node_state and new_state). These places are translated into nesC variables, and the variable type is determined by the colour of the CPN place. Line 9 corresponds to storing the incoming packet in the invocation of the event (ReceiveDIO), Line 10 to the external interface call and variable assignment of GetState, Line 11 to the variable assignment of the result of the `objectiveFunction` (ComputeNewState), and Line 12 corresponds to the call of the State interface (SetState) with the updated result from the `objectiveFunction`.

```
1    NetNode objectiveFunction(Nodes n, Nodes n2, Rank rank, DodagVerNum
         ver, Rank rrank, DodagVerNum rver, Nodes parent, STATE state) {
2        // return <TYPE: NetNode>;
3    }
4    event void DISDIO.receiveDIO(NodexPacket var_nodexpacket) {
5        NetNode new_state;
6        NetNode node_state;
7        NodexPacket packet;
8
9        packet = var_nodexpacket;
10       node_state = call State.getState();
11       new_state = objectiveFunction(...);
12       call State.setState(new_state);
13   }
```

Figure 4.16: Generated Behaviour of the receiveDIO Event

4.7 Discussion

By following the five refinement steps that we have identified, we can use the resulting refined CPN model to generate the structure of a nesC network protocol application and the internal behaviour of the protocol. The generated behaviour does not include the translation of CPN ML functions to nesC code, or branching of the control flow (if statements, and loops). There exists a tool for translating Standard ML to C[19], and automatically translating CPN ML to nesC has not been an area of focus in this thesis. A complete overview of the code generated from our CPN Roll Protocol Model can be found in Appendix C. Table 4.1 list the pragmatics used by the code generator to recognize structure and behaviour in CPN models.

Pragmatic	**Scope**	**Summary**
`<<component>>`	structure	Describes the relationship between TinyOS components and CPN submodules. We use the parameters of the pragmatic to differentiate between the five component classifications; boot, timed, external, dispatcher, and regular components.
`<<interface>>`	structure	Used together with arc annotation to describe the relationship between CPN places and TinyOS interfaces.
`<<command>>`	component	Describes the relationship between substitution transitions and TinyOS commands. Connected CPN places describe the return type, and the command parameters.
`<<event>>`	component	Describes the relationship between substitution transitions and TinyOS events. Connected CPN places describe the return type, and the event parameters.
`<<task>>`	component	Describes the relationship between substitution transitions and TinyOS tasks.
`<<id>>`	behaviour	Describes the control flow of an event, command or task. Used to create a clearly identifiable path of execution.
`<<invoke>>`	behaviour	Describes the invocation parameter types of commands and events in the control flow.
`<<var>>`	behaviour	Describes the relationship between CPN places and TinyOS variables. Used to store values between transitions in the control flow.

Table 4.1: List of Pragmatics

Chapter 5

Code Generation

In this chapter we introduce the technical details on how to generate code for the TinyOS platform. We have created a code generator that takes a CPN model, refined by the five steps in Chapter 4 as input, and generates code for the TinyOS platform. The TinyOS application is generated in two steps. The first step is to generate the structure of the application, and the second is to generate the behaviour of the TinyOS commands and events.

The application structure is generated from a CPN model refined by the steps we have shown in Chapter 4. The refined model include pragmatics that helps us by providing additional information to the model that we can derive patterns from. The reader will be introduced to how we use pragmatics to assist us in generating the application structure of a TinyOS network protocol, and how we have derived patterns to recognize CPN behaviour that can be translated to nesC code. By using the control flow pragmatics, we show that we are able to generate TinyOS nesC code corresponding to the behaviour of the CPN model.

5.1 The Code Generator

The code generator is written in Java and is using the Access/CPN framework for parsing the CPN model given as input. The code generator is split into two main parts: The generation of the TinyOS application structure and the generation of the internal behaviour of TinyOS events and commands. The code generator is built around a set of utilities for managing and mapping the relationship between the CPN model and code generated for the TinyOS platform. The code generator is structured as follows:

org.veiset.codegen makes up the main program of the code generator. This package is responsible for connecting the generated TinyOS application structure, and command and event behaviour together. Contains options for setting the output format of the generated code.

org.veiset.codegen.structure generates a Java representation for the outline of the TinyOS application structure. The package contains logic for creating and connecting interfaces and components together.

org.veiset.codegen.behaviour generates the behaviour of TinyOS commands and events based on the structure of CPN arcs, transitions and places that are annotated with pragmatics.

org.veiset.codegen.tinyos is used for representing TinyOS types, components, interface and wiring in Java. These Java representations also includes the translation of going from the Java representation and to corresponding TinyOS nesC code.

org.veiset.codegen.util contains utilities for mapping CPN concepts to Java representations (e.g, maps for looking up colour sets), tools for parsing pragmatics and CPN arc directions, and utilities for translating CPN colour set to TinyOS types.

5.2 TinyOS Application Structure

This section explains how we use the code generator and the refined model to outline the structure of a TinyOS network protocol application. The application structure consists of TinyOS components and interfaces, and these components are glued together with an application wiring file. The generated wiring specifies what component uses which interfaces, and which component is providing the used interfaces. To be able to wire components together, we need to define the signature of interfaces, and this is done by generating a nesC header file that maps CPN colour sets to nesC data structures.

5.2.1 Header file

The code generator we have developed, takes a CPN model as input, and based on the colour set declarations of the model, generates a list of corresponding nesC types. To be able to generate the nesC header file, we have to make some assumptions about the primitive types of both platforms.

We are able to generate nesC type representations of all colour sets by assuming that the relationship between primitives of nesC and CPN colour sets are as

described in Figure 5.1. The code generator requires that all colour sets used in the CPN model are based on the `UNIT`, `BOOL`, `INT` and `STRING` colour sets. By adding this restriction, we are able to use these as primitive types on both platforms, and this makes it possible to build structured colour sets that we can translate into corresponding nesC data structures.

```
1 colset UNIT = unit;       |      typedef int unit;
2 colset BOOL = bool;       |      typedef bool BOOL;
3 colset INT = int;         |      typedef int INT;
4 colset STRING = string;   |      typedef char string;
```

Figure 5.1: Assumptions of primitive header types

The colour sets are translated to nesC data structures by recursively checking if the type of the colour set is already defined. The colour set will be translated to the corresponding nesC data structure once all the colour sets it is built from are either previously defined or are of a primitive type. We categorize colour sets into five main types: Singletons, Lists, Enumerations, Products, and Unions. The code generator stores colour sets in each of these categories. When translating a type, the code generator will recursively lookup the type needed to define the new declaration.

The translation of singletons and lists is illustrated in Figure 5.2 by the Nodes and NodeList colour sets. We translate colour set consisting of a single colour to be generated as non-compound nesC data structures, and colour sets that are lists of non-compound data structure to be represented as nesC arrays. We set the default arrays size to 15, this is done by adding "`#define ARRAY_DEFAULT_SIZE 15`" to the top of the header file, which allows us to easily increase or decrease the default size. This constant is added to the size attribute of the generated array declarations, and to avoid multiple includes of the header file we use a header guard (`#ifndef GLOBAL_H_INCLUDED...`) to check if the header is previously included.

```
1 colset Nodes = int;       |      typedef int Nodes;
2 colset NodeList =         |      typedef Nodes
3            list Nodes;    |              NodeList[ARRAY_DEFAULT_SIZE];
```

Figure 5.2: Translation of variables and arrays

In CPNs, we have colour set that uses the `with` keyword, and these colour sets lists a set of constants. We translate these colour sets to nesC enumerations. This is done by iterating through the choices in the enumerate colour set and adding a increasing number value to the nesC `enum` representing. Figure 5.3 shows the step of going from a colour set with named values and to a nesC enumerations.

```
colset STATE = with          |     typedef enum {
      NODE                   |         NODE = 0,
    | ROOT                   |         ROOT = 1,
    | JOINED                 |         JOINED = 2,
    | JOINING                |         JOINING = 3,
    | ROOTJOINED             |         ROOTJOINED = 4,
    | WAITING                |         WAITING = 5,
    | INITROOT               |         INITROOT = 6,
    | INITNODE;              |         INITNODE = 7,
                             |     } STATE;
```

Figure 5.3: Translation of enums

CPN products are colour sets that consists of two or more colour sets, and these are represented as nesC `structs` by our code generator. The code generator goes through the colour sets constructing the product type, type by type, and creates a `struct` with a type and a name corresponding to each element in the CPN product. The `struct` attribute type is given by looking up the CPN type, and defining it if needed. The attribute name is given as the type of the attribute in lowercase, plus a `_val` postfix. The postfix is given so we avoid using type definitions as names. If multiple occurrences of the same colour set is defined in a single CPN product, we append a number to the postfix to make sure we have unique identifiable attribute names in the corresponding TinyOS `struct`. The translation of a CPN product to a nesC `struct` is shown in Figure 5.4.

```
colset NetNode = product     |     typedef struct {
      Nodes                  |         Nodes nodes_val;
    * Rank                   |         Rank rank_val;
    * DodagVerNum            |         DodagVerNum dodagvernum_val;
    * Nodes                  |         Nodes nodes_val2;
    * State;                 |         STATE state_val;
                             |     } NetNode;
```

Figure 5.4: Translation of products (data structures)

The CPN union type can contain a set of predefined colour sets and/or constants. We have translated the CPN unions into nesC unions. The translation is similar to that of products and nesC structs, with the exception of the named attribute. We have chosen to translate CPN record entries with the null-type (constants) into nesC `chars`. Figure 5.5 shows the `Dest` CPN union and the corresponding TinyOS union.

By introducing some basic CPN colour set and defining how they should be

```
colset Dest = union       |     typedef union {
    ALL: null             |         char ALL;
  + DEST: Nodes;          |         Nodes DEST;
                          |     } Dest;
```

Figure 5.5: Translation of unions

translated, we are able to generate a complete nesC header file that contains nesC data structures representing all the colour sets defined in a CPN model.

5.2.2 Interfaces

To be able to generate TinyOS interface signatures we annotate CPN places with the `<<interface>>` pragmatic, and inscribe the arcs going in and out of these places with additional information. This information specifies whether the interface is provided or used by the component (substitution transition) connected to the place. We look at the arcs inscribed with the `provides` keyword to find components providing interfaces. The components providing an interface contain submodules representing TinyOS events and commands, and based on these submodules we generate the signature of the interface.

Figure 5.6 shows three places annotated with the `<<interface>>` pragmatic connected to a substitution transition annotated with the `<<component>>` pragmatic. The interface DISDIO is used by the component. The code generator will check the component that uses the interface, and extract the signatures of the commands and events based on the submodules.

The commands and events will be added to the interface, allowing the code generator to determine the signature of the interface. The code generator will go through each place annotated with the `<<interface>` pragmatic and resolve the signature.

By looking at the DISDIO interface in Figure 5.6, the code generator will identify that the DISDIO component is using the interface. The code generator will then check the internal structure of the DISDIO substitution transitions for submodules annotated with the `<<event>>` or `<<command>>` pragmatic. These will be used to generate code describing the return type, the name, and the input parameters of the event or command. DISDIO contains two events, and Figure 5.7 shows the resulting code of the pattern shown in Figure 5.6.

By looking at places annotated with `<<interface>>`, their arcs, and the connected substitution transitions annotated with `<<component>>` we are able to generate TinyOS interfaces with signatures for commands and events.

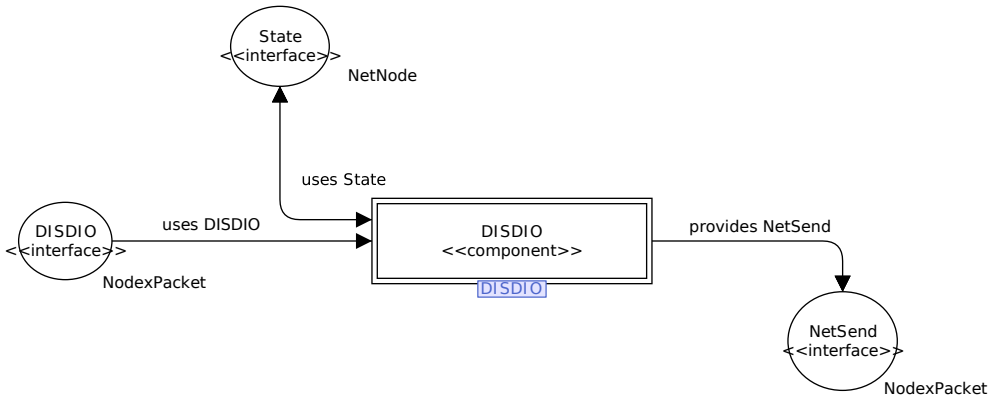


Figure 5.6: Interface Component Pragmatic

```
1 #include "global.h"
2 interface DISDIO {
3     event void receiveDIO(NodexPacket nodexpacket);
4     event void receiveDIS(NodexPacket nodexpacket);
5 }
```

Figure 5.7: Code for the Generated Interface

5.2.3 Components

Components are generated by looking for substitution transitions annotated with the `<<component>>` pragmatic. The code generator will look at the interfaces provided and used by the component, and add these to the module section of the TinyOS component. The substitution transitions have internal submodules that represent TinyOS events and commands. The internal submodules annotated with the `<<event>>` are generated as TinyOS events, and submodules annotated with `<<command>>` as TinyOS commands. The code generated for the TinyOS components consists of signatures corresponding to these events and commands.

We have derived multiple component types. Components annotated with `<<component (timed)>>` consist of submodules annotated with the `<<task>>` pragmatic, while the internal structure of the components annotated with the `<<component (external)>>` will not be generated.

Figure 5.8 shows a set of the components defined in the refinement steps. The components are annotated with the `<<component>>` annotation, which will allow the code generator to find them in the CPN model and generate code for the TinyOS platform.

Figure 5.9 shows the generated code for the DISDIO component. Line 1-5 describes

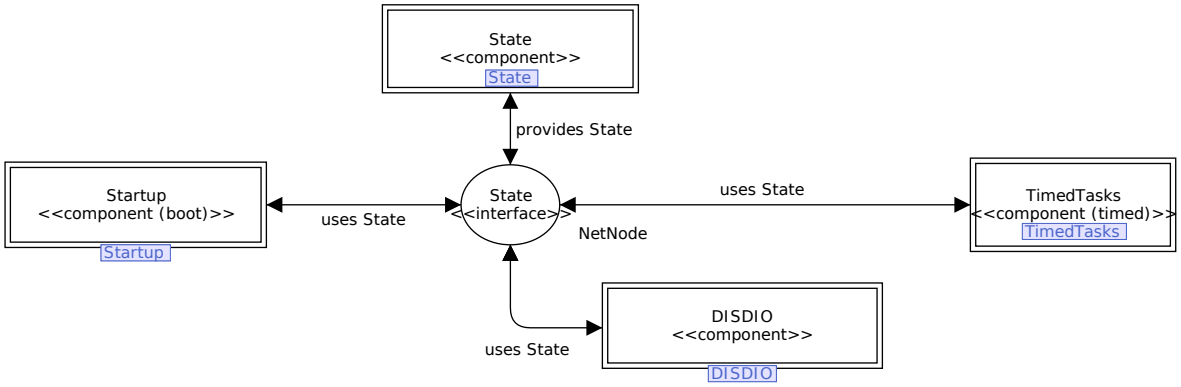


Figure 5.8: Component Patterns

the relationship between the component and the interfaces, and Line 6-9 describes the internal structure of the component.

```
1 module DISDIOC {
2     provides interface DISDIO;
3     uses interface State;
4     uses interface NetSend;
5 }
6 implementation {
7     event void DISDIO.receiveDIO(NodexPacket var_nodexpacket) { ... }
8     event void DISDIO.receiveDIS(NodexPacket var_nodexpacket) { ... }
9 }
```

Figure 5.9: Generated TinyOS component

5.2.4 Wiring

By looking at the relationship between the generated interfaces and components, we are able to generate the TinyOS application wiring. The first step of generating the wiring is done by listing the names of the substitution transitions annotated with the `<<component>>` pragmatic. The second step is done by looking at all the used interfaces, and these will then be mapped against the component proving them. Figure 5.10 shows the generated TinyOS application wiring for the CPN Roll Protocol model. Line 3 lists the components in the application, and Line 5-12 shows how the components are using interfaces, and which components that are providing the used interfaces, i.e, the wiring of the application.

```
configuration ConfigurationApp {}
implementation {
    components DISDIOC, StartupC, DAOC, DAOACKC, StateC, TimedTasksC,
        NetSendC;

    DISDIOC.State -> StateC.State;
    DAOC.State -> StateC.State;
    DAOACKC.State -> StateC.State;
    TimedTasksC.State -> StateC.State;
    StartupC.State -> StateC.State;
    NetSendC.NetSend -> DISDIOC.NetSend;
    NetSendC.NetSend -> DAO.NetSend;
    NetSendC.NetSend -> TimedTasks.NetSend;
}
```

Figure 5.10: Generated TinyOS wiring

5.3 Behaviour

Generating concrete behaviour from abstract models requires us to add restrictions or additional information to the model. By refining the CPN submodules representing the internal behaviour of the event and commands, we are able to get enough details to generate nesC code that reflects this behaviour. This is done by using pragmatics linked to control flow elements that makes it possible to clearly identify the path of execution. By structuring the CPN models of components and events based on a control flow and by using pragmatics, we are able to generate the nesC code that corresponds to the behaviour of the CPN submodule.

By looking closer at each step in the control flow, we have created distinguishable patterns for detecting behaviour that we could translate into nesC code. A pattern is a generalisation of a set of transitions, places and text strings. We can use the pattern to find concrete implementations of the general pattern in CPN models. To be able to generate behaviour for the TinyOS platform, we use a `<<ID>>` pragmatic for describing the control flow of a CPN model.

We create a graph representation of the control flow by using the CPN place annotated with `<<ID>>` that has an initial token as the source node of the graph. The source node is found up by looking up the control flow node that has an initial colour set token defined. We follow the connected transitions to the initial place and find the transitions leading to new control flow nodes. Transitions between two control flow nodes will be represented as an edge in the graph. The control flow of Figure 5.11 will be represented as the following graph:

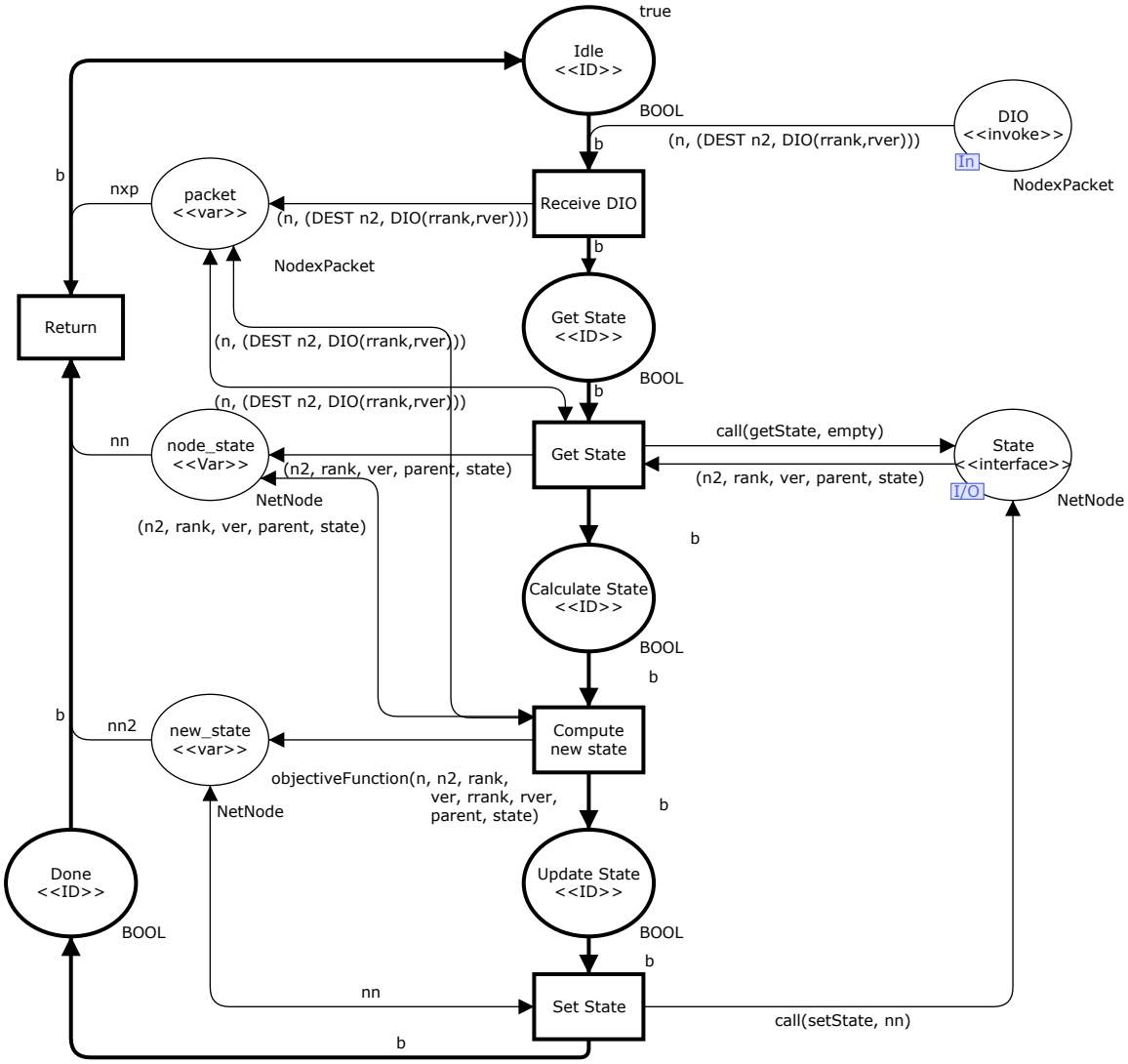


Figure 5.11: Receive DIO model behaviour

NODE	EDGE		NODE
Idle (initial)	(Receive DIO	->)	GetState
Get State	(Get State	->)	Calculate State
Calculate State	(Compute New State	->)	Update State
Update State	(Update State	->)	Done
Done	(Return	->)	Idle

By looking at each edge in the control flow, we are able to generate an outline of the nesC code corresponding to the CPN behaviour. We can add further details about each step by introducing patterns and conventions for describing behaviour.

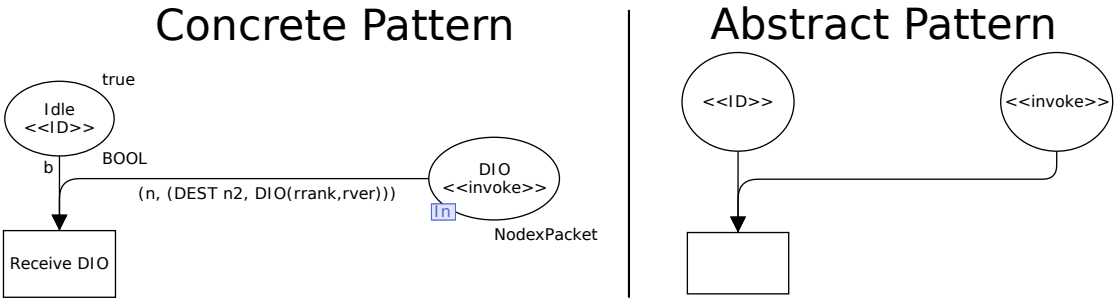
We introduce two pragmatics to describe details in the control flow. The `<<var>>` pragmatic to represent nesC variables, and the `<<invoke>>` pragmatic to represent nesC method invocation.

5.3.1 Method Invocation Pattern

There are two types of TinyOS event and command invocations that we wish to differentiate between. One is invocation with parameters, the other is without parameters. We have created a Method Invocation Pattern to find the entry point of TinyOS events and commands. We use the `<<ID>>` pragmatic to find the control flow node with an initial marking, and we use the `<<invoke>>` pragmatic, if it exists, to determine the parameter type.

The invoke pattern is detected by checking the transition between two control flow nodes. If the transition has an incoming arc connected to a place annotated with the `<<invoke>>` pragmatic, then we use the arc inscription and the colour set of the place to determine the signature of the nesC command or event.

The Concrete Pattern of Figure 5.12 is a concrete representation of the Method Invocation Pattern in the CPN Roll Protocol model, and shows the first transition of the control flow in Figure 5.11. The transition is connected to a place annotated with the `<<invoke>>` pragmatic, indicating that the event takes a parameter of the type NodexPacket. The Abstract Pattern of Figure 5.12 shows the general pattern that will match the Method Invocation Pattern in the code generator. This pattern can be combined with other patterns, and will be the first pattern to be matched and generated.



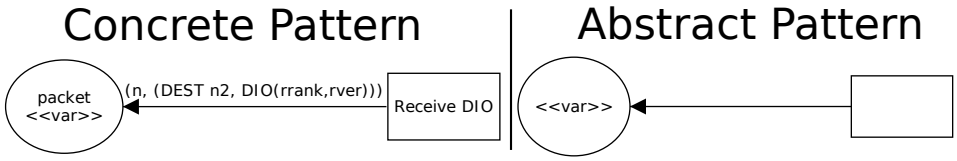

```
1 // TOStype return iface.method(inputType inputName)
2 event void DISDIO.receiveDIO(NodexPacket var_nodexpacket) {
```

Figure 5.12: Method Invocation Pattern

5.3.2 Assign Variable Pattern

To make data persistent over multiple steps in the execution of nesC events and commands, a correspondence between nesC variables and elements in the CPN model is needed. The Assign Variable Pattern is used for representing data storage, and the CPN tokens will be stored at places annotated with the `<<var>>` pragmatic.

Figure 5.13 shows how the Assign Variable Pattern is matched, and the code generated from the ReceiveDIO transition from Figure 5.11. The assign variable pattern is detected by checking that a transition has an arc going to a place annotated with the `<<var>>` pragmatic. This pattern will only be matched if the arc is going from a transition to a place. This pattern will not be matched if the arc is bidirectional.



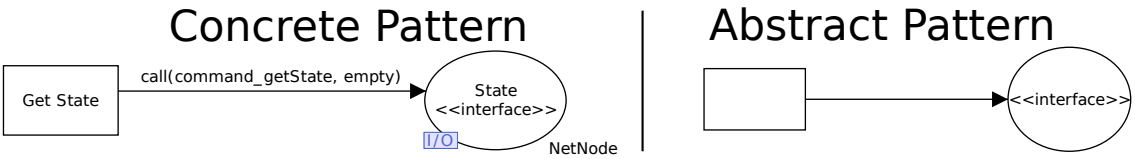

```
1 // var = value;
2 packet = var_nodexpacket;
```

Figure 5.13: Assign Variable Pattern

5.3.3 Interface Invoke Pattern

The Interface Invoke Pattern provides a way to describe TinyOS interface interaction in CPN models. We have introduced two CPN ML functions. A function for describing TinyOS commands, `call(commandName, parameter)`, and a function for describing events, `signal(eventName, parameter)`. The first argument is the name of the interface command or event, and the second argument is the parameter. The parameter can be `empty` to indicate that the interface invocation takes no parameters.

Figure 5.14 shows an example of the Interface Invoke Pattern of a command. The `getState` command is invoked without a parameter, and the Interface Invoke Pattern is matched if the transition have an outgoing arc inscribed with the `call` or `signal` ML function. The transition has to have an outgoing arc that is connected to a place annotated with the `<<interface>>` pragmatic. The Interface Invoke Pattern will be matched for both TinyOS commands and events.

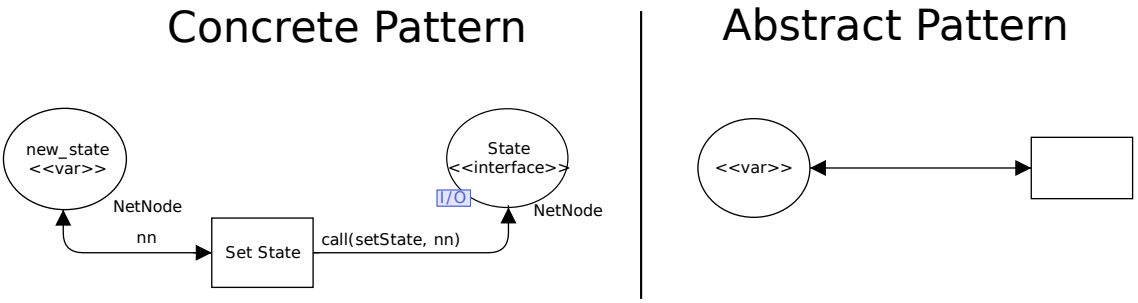



```
// call/signal interface.method(param);
call State.getState();
```

Figure 5.14: Call Interface Command pattern

5.3.4 Variable Usage Pattern

To utilize the data stored in places annotated with the `<<var>>` pragmatic, we need to introduce a pattern for using the CPN variable representations. The Variable Usage Pattern is not useful by itself, but by combining it with other patterns such as the Interface Invoke Pattern it allows us to generate interface calls with stored variables as parameters.




```
call State.setState(new_state);
```

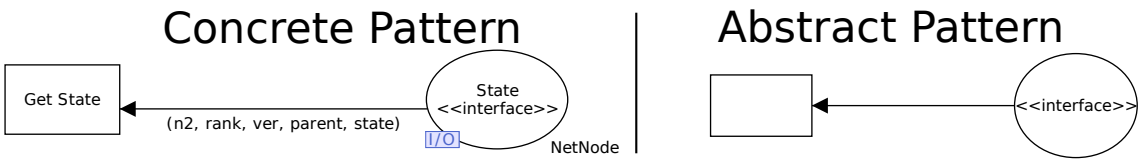
Figure 5.15: Use Variable Pattern

The Variable Usage Pattern is matched if the transition has an bidirectional arc going into a place annotated with the `<<var>>` pragmatic. Figure 5.15 shows how we can combine an interface call with the Variable Usage Pattern to use a stored value as the parameter of the interface call.

5.3.5 Interface Return Pattern

TinyOS interface calls can return values, and we need a way of representing this in the CPN model. By using the Interface Invoke Pattern, we can find interaction with interfaces. By checking for arcs going out from a place annotated with the `<<interface>>` pragmatic and into a transition, we can find interface interactions

that return values. The return type is determined by the colour set of the place annotated with the `<<interface>>` pragmatic. The Interface Return Pattern can be used together with the Assign Variable Pattern to store the returning value.




```
1 node_state = call State.getState();
```

Figure 5.16: Interface Return Pattern

Figure 5.16 shows how the Interface Return Pattern is matched. The Interface Return Pattern is matched by checking for arcs going from a place annotated with the `<<interface>>` pragmatic and to a transition in the control flow. By combining the interface invoke pattern, the interface return pattern, and the variable assignment pattern, we can store the returning value of an interface call to a variable.

The recognition of the patterns described in this chapter is implemented in the code generator tool we have developed. The installation and usage of the code generator is described in Appendix D.

Chapter 6

Application to Roll

In this chapter we look at the steps that are needed to go from the generated code and to a working network protocol application for the TinyOS platform. We show how we have taken the application generated by the code generator described in Chapter 5, and implement a subset of the Roll Protocol based on the generated nesC code.

6.1 Implementing Network Handlers

To be able to have a working application, we need to implement components for handling network packets. We organize the generated application in such way that we have one component for receiving, and one for sending. The component for receiving network packets has the responsibility of parsing and passing the packets to the right components.

TinyOS has a set of components that are used for managing network packet protocols[6]. We use the `Receive` interface provided by the `AMReceiveC` component to receive network packets from neighbouring nodes in the network, and we use the `AMSend` interface of the `AMSenderC` component to broadcast packets to the network.

6.1.1 The Dispatcher Component

The first step of going from the generated code to a runnable application is to create the component handling incoming network packets. The dispatcher component is based on the substitution transition Dispatcher, which is annotated with the `<<component (dispatcher)>>` pragmatic in the refined model (Figure 4.14).

The dispatcher receives and parses network packets that are represented as the `message_t*` TinyOS data structure (Chapter 3.2.1). Based on the result from parsing the packet, the dispatcher will pass the network packet as a `NodexPacket` data structure to the component that should further process it. The dispatcher uses the `Receive` interface, and has to implement the `receive` event.

Figure 6.1 illustrates how the dispatcher can be implemented. Line 1-2 describes the signature of the `Receive.receive` event, and Line 4-7 illustrate (in pseudo-code) how network packets can be passed from the dispatcher and to other components in the application.

```
1 event message_t* Receive.receive(
2        message_t* msg, void* payload, uint8_t l) {
3
4    NodexPacket packet;
5    ...
6    if (...) { signal DAOACK.receiveDAOACK(packet); };
7    else if (...) { signal DAO.receiveDAO(packet); };
8 }
```

Figure 6.1: AMReceive component - event for receiving network packets

Figure 6.2 shows how we create an instance of the generic `AMReceiveC` component with an alias in the application configuration file. We use the alias to connect the provided `receive` interface to the dispatch component.

```
1 component new AMReceiverC(1) as AMRecC;
2 DispatcherC.Receive -> AMRecC.Receive;
```

Figure 6.2: AMReceive component wiring

6.1.2 The NetSend Component

The second step is to implement the `NetSendC` component which is used by other components in the application for broadcasting network packets. The outline of the `NetSendC` component is generated from the substitution transition annotated with the `<<component (external)>>` pragmatic in the refined model. The `NetSendC` component is implementing the `AMSend` interface provided by the `AMSenderC` component to broadcast packets. The `AMSenderC` component comes with interfaces for managing and sending radio packets. We create an instance of the `AMSenderC` component, and wire the provided `AMSend` interface to the local `AMSend` interface of the `NetSend` component as shown by Figure 6.3. The `AMSend`

interface requires the component using the interface to implement a callback (`sendDone`) for the `send` command. We do not currently check for errors when sending packets, and the callback for sending packets is left unimplemented.

```
1 component new AMSenderC(1) as AMSendC;
2 NetSend.AMSend -> AMSendC.AMSend;
```

Figure 6.3: AMSender component wiring

The `NetSendC` component implements the `NetSend` interface (Figure 6.4), which provides components (DIO, DAO and DISDIO in the refined model) with the possibility of sending packets over the network. This is done by parsing the colour set representation of the `NodexPacket` and extracting the required attributes for creating a `message_t` data structure that can be sent through the `AMSend` interface.

```
1 interface NetSend {
2     event void netsend(NodexPacket packet);
3 }
```

Figure 6.4: NetSend interface

Through the implementation of the `NetSendC` we allow other components to signal the `NetSend.netsend` event, which calls the `send` command of the `AMSend` interface. Figure 6.5 shows the outline of the `NetSendC` component. Line 1-7 illustrate the `netsend` event that transform the data structure used internally between the components to the packet structure used by the `AMSend` interface for sending packets on the TinyOS platform. Line 8-10 show the unimplemented callback of the `AMSend.send` command.

```
1  event void NetSend.netsend(NodexPacket packet) {
2      message_t* msg;
3      am_addr_t addr;
4      uint8_t len;
5      ...
6      call AMSend.send(addr, msg, len);
7  }
8  event void AMSend.sendDone(message_t* msg, error_t error) {
9      // unimplemented callback
10 }
```

Figure 6.5: Implementation of the NetSendC component

By implementing the network handlers, we are able to receive and send network packets. The dispatcher receives the packets, parses them, and passes them on to the right components. The components then process the packets, and based on the payload execute logic. The components are then able to send responses based on the packets through the `NetSend` interface of the `NetSendC` component.

6.2 Implement interfaces for Timed Tasks

The code generator generates an outline of the `TimedTaskC` component. The `TimedTaskC` component contain tasks that should be execute periodically. We can implement the periodical execution of events by using the TinyOS `Timer` interface. We create an instance of the `Timer` interface for each of the timed tasks in the application. The timer have an event that will be executed when the timer is triggered, and this event will post a task to the TinyOS task queue.

To be able to use multiple instances of the Timer interface we give the instance of the Timer interface unique names in the configuration file, and bind them to the local interfaces used in the TimedTaskC component. This is illustrated in Figure 6.6.

```
1 components TimedTasksC;
2 components new TimerMilliC() as Timer0;
3 TimedTasksC.SendDISReqTimer -> Timer0;
4 TimedTasksC.Boot -> MainC.Boot;
```

Figure 6.6: Wiring for the TimedTask component

Figure 6.7 outlines how the `SendDISReq` task can be implemented using timers. To initialise the timers we call the `startPeriodic` command of the implemented `Timer` interface. The timers will be configured to execute at an given interval, and Line 7 shows that the SendDISReqTimer is executed every 10000ms (10 seconds). When the timer is executed by the TinyOS scheduler the `SendDISReqTimer.fired` event of Line 9 will be triggered, and this will in turn post the `SendDISReq` task to the TinyOS job queue.

By implementing timers for the `TimedTasksC` component, we are able to execute tasks at regular intervals, and this allow us to implement tasks that are periodically triggered.

```
module TimedTasksC {
    uses interface Boot;
    uses interface Timer<TMilli> as SendDISReqTimer;
}
implementation {
    event void Boot.booted() {
        call SendDISReqTimer.startPeriodic(10000);
    }
    event void SendDISReqTimer.fired() {
        post SendDISReq();
    }
    task void SendDISReq() {
        NodexPacket disc_pack;
        signal NetSend.netsend(disc_pack);
    }
}
```

Figure 6.7: Sample implementation of the TimedTask component

6.3 Porting functions

Some of the functionality of the CPN model is described as CPN ML functions, and these have to be manually translated to nesC code. To get a working implementation of the generated application, we have to translate all the used Standard ML functions to corresponding nesC code.

Figure 6.8 shows the CPN ML function for determining if a node is part of a *DODAG*, and Figure 6.9 shows the manually translated nesC version of this function.

```
fun noDodag(parent, rrank, rver) =
    if ((rver=0) andalso (rrank=0) andalso (parent=0))
        then true
    else false;
```

Figure 6.8: CPN ML implementation of the noDodag function

```
BOOL noDodag(int parent, int rrank, int rver) {
    if (rver == 0 && rrank == 0 && parent == 0) {
        return 1;
    }
    return 0;
}
```

Figure 6.9: nesC implementation of the noDodag function

6.4 TOSSIM

To be able to test the application locally, we need to have an environment to simulate nodes in a network. This is done by using the TOSSIM python framework, which allows us to interact with the generated application. The TOSSIM framework allow us to configure the network topology where we can describe the connectivity between the nodes. The first step of simulating a TinyOS application is to is to initialize the TOSSIM environment and the radio links.

```
tossim = Tossim([])
radio = t.radio()
```

The second step is defining the simulation scenario. This is done by defining the topology, where we can add links between nodes and describe the connectivity for each link. Figure 6.10 shows how we can a get a representation of the topology by reading a scenario from file.

```
with open("topology.txt") as toplogy:
    for link in toplogy:
        node1, node2, db = link.split()
        radio.add(int(node1), int(node2), float(db))
```

Figure 6.10: Setting up the TOSSIM Topology

To make the simulation realistic we can add a random noise to the network by sampling noise from an actual field study. By doing this we can simulate how the physical environment (e.g, changes in temperature, weather conditions, etc) can have an impact on how the network protocol behaves. Figure 6.11 shows how we add noise to the TOSSIM network model.

TOSSIM allow us to simulate the booting sequence of nodes running the TinyOS application. After defining the topology, the noise models and the booting sequences, we are able to simulate network nodes in the TOSSIM TinyOS environment. We simulate events in the environment by calling the `runNextEvent()`

```
with open("sample-noise.txt", "r") as noise:
    for sample in noise:
        for node in nodes:
            tossim.getNode(node).addNoiseTraceReading(int(sample))

for node in nodes:
    tossim.getNode(node).createNoiseModel()
```

Figure 6.11: Creating noise for the TOSSIM environment

method of the TOSSIM environment. The installation and setup of TOSSIM is described in Appendix A.

```
tossim.getNode(1).bootAtTime(1800009)
tossim.getNode(2).bootAtTime(2000002)
tossim.getNode(3).bootAtTime(8008135)
for _ in range(60): tossim.runNextEvent()
```

Figure 6.12: Simulating the behaviour of nodes in the network

The generate code outlines the behaviour of the functions, and we are able to generate most of the application. Going from the generated code and to a working implementation of the network protocol, we have to translate the functions used in the CPN model to nesC functions, implement network handlers and implement the timed tasks. The generated code constitute most of the application, and we see that that most of the manual implementation of the network handlers can be reused for other protocols.

Chapter 7

Conclusions and Future Work

In this thesis we have developed and applied a method for refining CPN network protocol models, a refinement process that results in a model we can use for automatically generating code for a corresponding TinyOS network protocol application. We have created five distinct refinement steps that is used in the process of refining a CPN model. We refine the model by adding explicit details to describe the relationship to the TinyOS platform, and by restructuring the CPN model to better reflect the structure of the TinyOS platform.

We have explored the possibilities of extending the use of pragmatics, and we have created pragmatics for describing the relationship between CPN concepts and TinyOS components, interfaces, events and commands. We have also created a set of pragmatics for describing the internal behaviour of events and commands.

We have created a code generator that uses these pragmatics to generate an outline of the structure and behaviour of a TinyOS application. We have used pragmatics in our case study of the Roll Protocol to map the structure and behaviour of the CPN model to a corresponding TinyOS network protocol application. We have been able to show a relationship between concepts in CPN models and variables, method invocations, and the control flow of commands and events in TinyOS applications.

We will discuss the results and findings of the model refinement process, the code generator, the generated code, and the manual implementation needed to run the application on the target platform in greater detail in the sections that follow.

7.1 Model Refinement Process

The model refinement shows that we can take a platform independent model and by refining it accordingly to the refinement steps we have derived, end up with a model that we can use for generating platform specific code.

We have successfully used pragmatics to describe the relationship between an abstract CPN model and the TinyOS platform. We have used pragmatics to create patterns that can be detected by a code generator, and these patterns are used to relate CPN concepts to TinyOS structure and behaviour. The original CPN model was modelled very compactly, and described the behaviour with a very small set of arcs, transitions and places. This made it hard to clearly identify the execution flow, and by restructuring the model using a control flow pattern, we have been able to clearly identify the path of execution. This enabled us to reason about the corresponding TinyOS behaviour. The CPN model is refined in such a way that it allow us to generate the structure and the internal behaviour of the corresponding TinyOS network protocol application.

7.2 Code Generation

The code generator we have created takes a refined CPN model as input, and generates nesC code for the TinyOS platform. We have looked closer at how we can recognize structural patterns, and how we can use these patterns to describe the relationship between a CPN model and the corresponding TinyOS application.

To be able to generate code based on recognizable patterns, we have annotated the CPN arcs, substitution transitions and places with pragmatics. By using pragmatics and changing the structure of the model, we are able to generate nesC code corresponding to TinyOS wiring, components, interfaces, and headers. The generated header file contains the translation of all the CPN colour sets, which are represented as TinyOS data structures.

We have restructured the behaviour of the CPN model to describe a control flow. By representing the control flow as a graph, we can identify each step of the execution, and translate it to TinyOS platform specific behaviour. Using predefined patterns we are able to generate behaviour for method invocation, variable assignments, interface invocations, interface invocations with variables as parameters and assigning interface return values to variables.

The case study in this thesis showed that by using a code generator that detects structural patterns in CPN models, we are able to generate human readable TinyOS application code. One of the benefits of using a structure-based code generator is

the readability of the generated code, and when giving descriptive names to the CPN colour sets, places and transitions, we end up with generated code that is very readable. The readability of the generated code makes the application easy to modify. The current version of the code generator is not able to generate a complete representation of the CPN model behaviour, and some manual work is needed for going from generated application to an operational implementation of the network protocol. To get a runnable application we have to implement the network dispatcher receiving the packets, the interface for broadcasting packets to the network, and the timers for executing tasks periodically.

Even though the code generator does not currently generating a complete representation of the TinyOS Roll Protocol application, we can argue for the importance of generating readable code. The generated code is very readable, making the application easy to extend and change. One unresolved issue with the manual implementation process, is ensuring the correctness of going from the generated network protocol application to a runnable implementation of it.

7.3 Future Work

Based on the results obtained in this thesis, there are several areas where the refinement process and the code generation of TinyOS applications could be explored further.

7.3.1 Automated Testing and Analysis

An area that would have been especially interesting to explore further is the possibility of automatic test generation. It would also be interesting to look further into testing to ensure the correctness of the CPN model, and automatic validation of the correctness for each step of the refinement.

By looking into the generation of automatic test cases for the TOSSIM simulation tool, it should be possible to generate test cases for the behaviour of the CPN model that reflects what the behaviour of the completed implementation of the generated TinyOS application should be. It might be possible to make a test framework that generates network test cases, runs them through both CPN and TOSSIM and matching the result of both the platforms. This would give us a way of ensuring that the result of the manual implementation is correct and matches the behaviour of the CPN model.

As an exercise of this it would have been interesting to rigorously test the CPN model by simulating it and match it against the result of the generated TinyOS application. We could use state space analysis tools to ensure correctness of

the CPN model, and possibly use the result of the space analysis to reason about the generated code. This would give us more confidence in our generated implementation being correct.

Another aspect that would be interesting to look at is the power consumption of the generated code. We could run the generated code through the nesc2cpn[7] program to generate a CPN model, which would allow us to monitor the power consumption of the generated TinyOS application. By doing this we could reason about the efficiency of the generated code, and if needed, optimize it for efficiency. It would be interesting to use the result of this and compare it to the power consumption of a manual implementation of the protocol.

7.3.2 Improving the Code Generator

The current version of the code generation is a proof of concept and is missing some features that would have been nice to have in a final product. Currently, the code generator does not generate the dispatcher, does not include branching of the control flow and is only able to bind one variable in each step of the control flow. These limitations with the code generator has not been implemented due to time constraints. There is no technical reason that prohibits the implementation of these features.

By looking closer at the process of going from the application to the Roll Protocol implementation, we see that some of the implementation steps that are done manually could possibly be automatically generated. The substitution transition representing the `dispatcher` in the refined model is structured in such a way that the enabling of transitions could be translated into switch statements. By looking closer at the Figure 4.13, we see that each of the arcs correspond to a statement, and these can be used to generate a switch statement over different cases that would correspond to passing network packets. The tasks in the `TimedTasks` component follow a structure that could possibly allow us to generate the timers and wiring for each of the tasks.

The code generator does not currently support branching in the control flow. Implementing branching of the control flow would enable us to describe loops and conditional statements. These features could be added by implemented a more advanced graph representing of the control flow, where a node with two or more outgoing edges in the graph would correspond to a conditional statement (if, else if, else). If the graph has a cycle, we would assume that this corresponds to a loop, and nodes with multiple incoming edges would represent the start of a loop in the control flow. The node representing the loop entry point should also contain the condition for when the loop should break.

The code generator does not currently ensure that the variables used in the

Variable Usage Pattern matches the signature of the interface, and as a result of this we do not support multiple variable assignments or usages in a single step of the control flow. This could possibly be solved by matching the types of the CPN arcs to the places annotated with the `<<var>>` pragmatic. One unresolved issue with this is how to handle the variable assignment if multiple variables of the same type are assigned in one transition. It would be interesting to look into this issue further and explore the possibilities of using multiple variable assignments and uses in a single step.

Currently the code generator will only work on a model after all the five refinement steps have been completed. It would have been nice if the code generator was able to outline the corresponding code for each of the steps in the refinement. The code generator could be used to guide the user to what the next steps in the refinement would be. This improvement to the code generator could also be used for checking for mistakes in the refinement steps (e.g, unknown pragmatics, duplicate annotations, etc.).

One possible improvement of the current code generator would be to have it automatically translate the CPN ML functions into nesC functions. Research has shown that is possible to translate Standard ML to C[19], and this indicates that translation of CPN ML functions to nesC functions should be possible.

In this thesis we have shown that we can refine a platform independent CPN model to better match the structure of a target platform. Refining the CPN Roll Protocol model has allowed us to generate platform specific code for the TinyOS platform. We have derived five general steps used in the refinement of CPN models, and we have successfully shown how we can use pragmatics to make a CPN model sufficiently detailed to use for code generation for the TinyOS platform.

List of Figures

Bibliography

[1] Routing over low power and lossy networks. https://datatracker.ietf.org/wg/roll/charter, May 2013.

[2] Tinyos - documentation wiki. http://docs.tinyos.net, 2013.

[3] Tinyos - documentation wiki. http://docs.tinyos.net/tinywiki/index.php/tinyos_network_protocol_working_group, 2013.

[4] Tinyos - enhancement proposals. http://docs.tinyos.net/tinywiki/index.php/teps, 2013.

[5] Tinyos - platform hardware. http://docs.tinyos.net/tinywiki/index.php/platform_hardware, 2013.

[6] Tinyos packet protocols. http://www.tinyos.net/tinyos-2.x/doc/html/tep116.html, May 2013.

[7] Antonio Damaso, Davi Freitas, Nelson Rosa, Bruno Silva, and Paulo Maciel. Evaluating the power consumption of wireless sensor network applications using models. *Sensors*, 13(3):3473–3500, 2013.

[8] Kristian Leth Espensen and Mads Keblov Kjeldsen. Automatic code generation from process-partitioned coloured petri net models. Master's thesis, Department of Computer Science, University of Aarhus, 2008.

[9] A. Conta et.al. Internet control message protocol (icmpv6) for the internet protocol version 6 (ipv6) specification (rfc 4443). Technical report, Network Working Group, IETF, 2006.

[10] Ed T. Winter et.al. Ipv6 routing protocol for low power and lossy networks. Technical report, The Internet Engineering Task Force, 2012.

[11] Kurt Jensen and Lars M. Kristensen. *Coloured Petri Nets: Modelling and Validation of Concurrent Systems*. Springer, 1 edition, 2009.

[12] S. KIm, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon.

Health monitoring of civil infrastructures using wireless sensor networks. *In IPSN '07: Proceedings of the Sixth International Conference on Information Processing in Sensor Networks*, 2007.

[13] Lars M. Kristensen, Peter Mechlenborg, Lin Zhang, Brice Mitchell, and Guy E. Gallasch. Model-based development of a course of action scheduling tool. *Int.J.Softw.Tools Technol.Transf.*, pages 10(1):5–14, 2007.

[14] Mads K. Kjeldsen Kristian L. Espensen and Lars M. Kristensen. Towards modelling and validation of the dymo routing protocol for mobile ad-hoc networks.

[15] Philip Levis. *TinyOS Programming*. 2006.

[16] J. Liu, N. Priyantha, F. Zhao, C.-J. M. Liang, and Q. Wang. Towards discovering data center genome using sensor nets. *In EmNets '08: Proceedings of the Fifth Workshop on Embedded Networked Sensors*, 2008.

[17] Kjeld Hoyer Mortensen. Automatic code generation method based on coloured petri net models applied on an access control system. *In Proceeding of ICATPN '00*, pages 367–386, 2000.

[18] Kent Inge Simonsen, Lars Michael Kristensen, and Ekkart Kindler. *Code Generation for Protocols from CPN models Annotated with Pragmatics*. IMM-Technical Report-2013. Technical University of Denmark, 2013.

[19] David Tarditi, Peter Lee, and Anurag Acharya. No assembly required: Compiling standard ml to c. Technical report, ACM Letters on Programming Languages and Systems, 1990.

[20] Ed. P. Thubert. Objective function zero for the routing protocol for low-power and lossy networks (rpl). Technical report, The Internet Engineering Task Force, 2012.

[21] M. Westergaard. Access/CPN 2.0: A High-Level Interface to CPN Models. In *Proc. of ICATPN'11*, volume 6709 of *LNCS*, pages 328–337. Springer, 2011.

Appendix A

Installing TinyOS and running nesC applications

A.1 Installing TinyOS

This install guide targets Debian based Linux distributions, and is based on the official install that can be found on `http://docs.tinyos.net/tinywiki/index.php/Installing_TinyOS`. The official guide also covers other platforms and operating systems.

Firstly we need to install a compatible Linux distribution. Xubuntu is a lightweight alternative. If we do not wish to install Linux natively we can run in a virtual environment, this can be done by install VirtualBox or VMWare. After the Linux distribution is set up we need to install the TinyOS packages. Add the TinyOS package repository and install it using the Ubuntu package-manager.

1 - **Add the source for the debian packages**

```
#/etc/apt/sources.list
deb http://tinyos.stanford.edu/tinyos/dists/ubuntu lucid main
```

2 - **Install the software**

```
sudo apt-get update
sudo apt-get install tinyos-2.1.*
```

3 - **Set up the bash environment to include the TinyOS environment**

```
export MAKERULES=/opt/tinyos-2.1.2/support/make/Makerules
export TOSDIR=/opt/tinyos-2.1.2/tos
```

4 - **Install the python development libraries**

```
sudo apt-get install python2.7-dev
```

A.2 Running nesC applications

1 - **Compile**

```
make micaz sim
```

2 - **Running simulation**

```
python sim.py # where sim.py is a python2 simulation case file
```

Appendix B

Roll Protocol nesC example

Makefile

```
COMPONENT=RPLProtocolAppC
include $(MAKERULES)
```

DataTypes.h

```
#ifndef DATATYPES_H_INCLUDED
#define DATATYPES_H_INCLUDED

typedef enum {
    DAOpack     = 1,
    DAOACKpack  = 2,
    DIOpack     = 3,
    DISpack     = 4,
} PacketType;


typedef enum {
    INITNODE   = 0,
    NODE       = 1,
    JOINING    = 2,
    JOINED     = 3,
    WAITING    = 4,
    ROOT       = 5,
    INITROOT   = 6,
    ROOTJOINED = 7,
} State;

typedef struct {
```

```
    uint8_t source;
    uint8_t dest;
    PacketType packetType;
} Packet;

typedef struct {
    uint8_t id;
    uint8_t rank;
    uint8_t dodagN;
    uint8_t parentId;
    State state;
} NetNode;

#endif
```

RPLProtocolAppC.nc

```
configuration RPLProtocolAppC { }
implementation {
    components MainC, RPLProtocolC;
    components DAOC;
    components DIOC;

    RPLProtocolC.Boot -> MainC.Boot;
    RPLProtocolC.DAO -> DAOC.RPLPacket;
    RPLProtocolC.DIO -> DIOC.RPLPacket;

    DAOC.DODAG -> RPLProtocolC.DODAG;

}
```

RPLProtocolC.nc

```
#include "DataTypes.h"

module RPLProtocolC {
    provides interface DODAG;

        uses interface Boot;
    uses interface RPLPacket as DAO;
    uses interface RPLPacket as DIO;
}

implementation {
    NetNode node;
```

```
13
14      command State DODAG.getState() { return node.state; }
15      command void DODAG.setState(State state) {
16          dbg("dbg", "%s RPL | changeState %i -> %i\n",
                sim_time_string(), node.state, state);
17          node.state = state;
18      }
19
20      event void Boot.booted() {
21          // debug
22          NetNode nnode = { .id = 1, .rank = 0, .dodagN = 0, .parentId =
                0, INITNODE };
23          Packet packet = { .source = 2, .dest = 1, .packetType = DAOpack
                };
24          dbg("dbg", "%s RPL | Application booted.\n", sim_time_string());
25
26          node = nnode;
27          call DAO.receive(packet);
28      }
29
30      event void DAO.send(Packet packet) {
31          dbg("dbg", "%s RPL | DAO.send(Packet packet)\n",
                sim_time_string());
32      }
33
34      event void DIO.send(Packet packet) {
35          dbg("dbg", "%s RPL | DAO.send(Packet packet)\n",
                sim_time_string());
36      }
37
38
39  }
```

DAOC.nc

```
1   #include "DataTypes.h"
2
3   module DAOC {
4       provides interface RPLPacket;
5       uses interface DODAG;
6   }
7
8   implementation{
9       command void RPLPacket.receive(Packet p) {
10          State state = call DODAG.getState();
```

```
        dbg("dbg", "%s DAO | RPLPacket.receive(..) with state %i\n",
            sim_time_string(), state);
        call DODAG.setState(JOINED);
        signal RPLPacket.send(p);
    }
}
```

DIOC.nc

```
#include "DataTypes.h"

module DIOC {
    provides interface RPLPacket;
}

implementation{
    command void RPLPacket.receive(Packet p) {
        dbg("Boot", "%s | RPLPacket.receive(..)\n", sim_time_string());
        signal RPLPacket.send(p);
    }
}
```

DODAG.nc

```
#include "DataTypes.h"

interface DODAG {

    command State getState();
    command void setState(State state);
}
```

RPLPacket.nc

```
#include "DataTypes.h"

interface RPLPacket {
    command void receive(Packet packet);
    event void send(Packet packet);
}
```

rplsim.py

```
from TOSSIM import *
import sys
```

```
3
4  log = open("log.txt","w")
5
6  t = Tossim([])
7  t.setTime(0)
8  t.addChannel("dbg", log)
9
10 m0 = t.getNode(0)
11 #m1 = t.getNode(1)
12 m0.bootAtTime(300)
13 #m1.bootAtTime(3000)
14
15 for _ in range(5):
16     t.runNextEvent()
```

Appendix C

Generated Code

```
// Code Generating for 'RollProtocol' at 25.05.2013 02:43:19

// Confugration file generated at 25.05.2013 02:43:19
configuration ConfigurationApp {}
implementation {
    components DISDIOC;
    components StartupC;
    components DAOC;
    components DAOACKC;
    components StateC;
    components TimedTasksC;
    components NetSendC;

    DISDIOC.State -> StateC.State;
    StartupC.State -> StateC.State;
    DAOC.State -> StateC.State;
    DAOACKC.State -> StateC.State;
    TimedTasksC.State -> StateC.State;
    NetSendC.NetSend -> DISDIOC.NetSend;
}

// Interface file 'State' generated at 25.05.2013 02:43:19
#include "global.h"
interface State {
    event void receiveDIO(NodexPacket nodexpacket);
    event void receiveDIS(NodexPacket nodexpacket);
}

// Interface file 'NetSend' generated at 25.05.2013 02:43:19
#include "global.h"
```

```
31 interface NetSend {
32 }
33
34 // Interface file 'DAOACK' generated at 25.05.2013 02:43:19
35 #include "global.h"
36 interface DAOACK {
37     event void receiveDAOACK(NodexPacket nodexpacket);
38 }
39
40 // Interface file 'DAO' generated at 25.05.2013 02:43:19
41 #include "global.h"
42 interface DAO {
43     event void receiveDAO(NodexPacket nodexpacket);
44     event void receiveDAOwACK(NodexPacket nodexpacket);
45 }
46
47 // Interface file 'DISDIO' generated at 25.05.2013 02:43:19
48 #include "global.h"
49 interface DISDIO {
50     event void receiveDIO(NodexPacket nodexpacket);
51     event void receiveDIS(NodexPacket nodexpacket);
52 }
53
54
55 // Component DISDIOC.nc at 25.05.2013 02:43:19
56 module DISDIOC {
57     provides interface NetSend;
58     uses interface State;
59     uses interface DISDIO;
60 }
61 implementation {
62
63     event void DISDIO.receiveDIO(NodexPacket var_nodexpacket) {
64         NetNode new_state;
65         NetNode node_state;
66         NodexPacket packet;
67
68         packet = var_nodexpacket;
69         node_state = call State.getState();
70         new_state = objectiveFunction(...);
71         call State.setState(new_state);
72     }
73
74     event void DISDIO.receiveDIS(NodexPacket var_nodexpacket) {
75         NetNode node_state;
```

```
76          NodexPacket packet;
77
78          packet = var_nodexpacket;
79          node_state = call State.getState();
80          signal NetSend.netsend(node_state);
81      }
82
83  }
84
85  // Component StartupC.nc at 25.05.2013 02:43:19
86  module StartupC {
87      uses interface State;
88  }
89  implementation {
90
91  }
92
93  // Component DAOC.nc at 25.05.2013 02:43:19
94  module DAOC {
95      provides interface NetSend;
96      uses interface State;
97      uses interface DAO;
98  }
99  implementation {
100
101     event void DAO.receiveDAO(NodexPacket var_nodexpacket) {
102         NodexPacket packet;
103
104         packet = var_nodexpacket;
105     }
106
107     event void DAO.receiveDAOwACK(NodexPacket var_nodexpacket) {
108         NetNode node_state;
109         NodexPacket packet;
110
111         packet = var_nodexpacket;
112         node_state = call State.getState();
113         signal NetSend.netsend(node_state);
114     }
115
116 }
117
118 // Component DAOACKC.nc at 25.05.2013 02:43:19
119 module DAOACKC {
120     uses interface State;
```

```
121     uses interface DAOACK;
122 }
123 implementation {
124 
125     event void DAOACK.receiveDAOACK(NodexPacket var_nodexpacket) {
126         NetNode new_state;
127         NetNode node_state;
128         NodexPacket packet;
129 
130         packet = var_nodexpacket;
131         node_state = call State.getState();
132     }
133 
134 }
135 
136 // Component StateC.nc at 25.05.2013 02:43:19
137 module StateC {
138     provides interface State;
139 }
140 implementation {
141 
142 }
143 
144 // Component TimedTasksC.nc at 25.05.2013 02:43:19
145 module TimedTasksC {
146     provides interface NetSend;
147     uses interface State;
148 }
149 implementation {
150 
151     task void <<unknown>>.SendDISReq() {
152         NodexPacket disc_pack;
153         NetNode node_state;
154 
155         node_state = call State.getState();
156         signal NetSend.netsend(disc_pack);
157     }
158 
159     task void <<unknown>>.SendDAO() {
160         NodexPacket disc_pack;
161         NetNode node_state;
162         INT enable_ack;
163 
164         call State.getState();
165         node_state = nn;
```

```
166         disc_pack = call State.setState(node_state);
167         call State.setState(node_state);
168     }
169 
170     task void <<unknown>>.IncreaseDODAG() {
171         NetNode node_state;
172 
173         node_state = call State.getState();
174         call State.setState(node_state);
175 
176     }
177 
178     task void <<unknown>>.Timeout() {
179         NetNode node_state;
180 
181         node_state = call State.getState();
182         call State.setState(node_state);
183     }
184 
185 }
186 
187 // Component NetSendC.nc at 25.05.2013 02:43:19
188 module NetSendC {
189     uses interface NetSend;
190 }
191 implementation {
192 
193 }
194 
195 // Header file 'global.h' generated at 25.05.2013 02:43:19
196 #ifndef GLOBAL_H_INCLUDED
197 #define GLOBAL_H_INCLUDED
198 
199 #define ARRAY_DEFAULT_SIZE 15
200 
201 typedef int unit;
202 typedef char string;
203 typedef int INT;
204 typedef INT DodagVerNum;
205 typedef unit UNIT;
206 typedef INT Rank;
207 typedef int Nodes;
208 typedef char null;
209 typedef char ALL;
210 typedef char DEST;
```

```
211 typedef union {
212     char ALL;
213     Nodes DEST;
214 } Dest;
215 typedef char DIS;
216 typedef string STRING;
217 typedef STRING Data;
218 typedef int Options;
219 typedef struct {
220      Rank rank_val;
221      DodagVerNum dodagvernum_val;
222      Data data_val;
223      Options options_val;
224 } DAOpack;
225 typedef char DAO;
226 typedef struct {
227      Rank rank_val;
228      DodagVerNum dodagvernum_val;
229 } DAOACKpack;
230 typedef char DAOACK;
231 typedef struct {
232      Rank rank_val;
233      DodagVerNum dodagvernum_val;
234 } DISResp;
235 typedef char DIO;
236 typedef union {
237     char DIS;
238     DAOpack DAO;
239     DAOACKpack DAOACK;
240     DISResp DIO;
241 } PacketType;
242 typedef struct {
243      Dest dest_val;
244      PacketType packettype_val;
245 } Packet;
246 typedef struct {
247      Nodes nodes_val;
248      Packet packet_val;
249 } NodexPacket;
250 typedef NodexPacket netReceive;
251 typedef char VOID;
252 typedef VOID param_getState;
253 typedef VOID return_netSend;
254 typedef enum {
255      NODE = 0,
```

```
256     ROOT = 1,
257     JOINED = 2,
258     JOINING = 3,
259     ROOTJOINED = 4,
260     WAITING = 5,
261     INITROOT = 6,
262     INITNODE = 7,
263 } STATE;
264 typedef STATE return_getState;
265 typedef bool BOOL;
266 typedef BOOL ConfigParam;
267 typedef INT event_booted;
268 typedef NodexPacket event_send;
269 typedef VOID return_receive;
270 typedef VOID return_setState;
271 typedef NodexPacket param_netSend;
272 typedef INT ID;
273 typedef event_booted interface_boot;
274 typedef Nodes NodeList[ARRAY_DEFAULT_SIZE];
275 typedef enum {
276     UP = 0,
277     DOWN = 1,
278 } LinkStatus;
279 typedef struct {
280     Nodes nodes_val;
281     Nodes nodes_val2;
282     LinkStatus linkstatus_val;
283 } NodexNodexLinkStatusxLinkType;
284 typedef NodexNodexLinkStatusxLinkType
       TopologyChanges[ARRAY_DEFAULT_SIZE];
285 typedef struct {
286     return_netSend return_netsend_val;
287     param_netSend param_netsend_val;
288 } netSend;
289 typedef union {
290    char netSend;
291    char netReceive;
292 } interface_Network;
293 typedef struct {
294     INT int_val;
295     STATE state_val;
296 } param_setState;
297 typedef struct {
298     return_setState return_setstate_val;
299     param_setState param_setstate_val;
```

```
300 } setState;
301 typedef struct {
302     return_getState return_getstate_val;
303     param_getState param_getstate_val;
304 } getState;
305 typedef union {
306    char setState;
307    char getState;
308 } interface_State;
309 typedef char receiveDIS;
310 typedef char receiveDIO;
311 typedef union {
312    char receiveDIS;
313    char receiveDIO;
314 } interface_DISDIO;
315 typedef struct {
316     NodexPacket nodexpacket_val;
317     Nodes nodes_val;
318 } param_receive;
319 typedef struct {
320     return_receive return_receive_val;
321     param_receive param_receive_val;
322 } command_receive;
323 typedef union {
324    char command_receive;
325    char event_send;
326 } interface_RPLPacket;
327 typedef char receiveDAO;
328 typedef char receiveDAOwACK;
329 typedef union {
330    char receiveDAO;
331    char receiveDAOwACK;
332 } interface_DAO;
333 typedef char netsend;
334 typedef union {
335    char netsend;
336 } interface_NetSend;
337 typedef char receiveDAOACK;
338 typedef union {
339    char receiveDAOACK;
340 } interface_DAOACK;
341 typedef struct {
342     Nodes nodes_val;
343     NodeList nodelist_val;
344 } Topology;
```

```
345 typedef struct {
346     Nodes nodes_val;
347     Rank rank_val;
348     DodagVerNum dodagvernum_val;
349     Nodes nodes_val2;
350     STATE state_val;
351 } NetNode;
352 #endif
```

Appendix D

Using the Code Generator

The executable Java file for the code generator can found online at `http://veiset.org/master/binaries/`.

The JAR contain the Eclipse Modelling Framework (EMF) and Access/CPN library used by the code generator. To run the code generator, we simply execute the JAR file with parameters for input and output.

```
java -jar codegen-1.0.jar /home/vz/models/refined.cpn /home/vz/gen/
```

The line above takes a CPN model as input (`/home/vz/models/refined.cpn`) and outputs the corresponding nesC code to the specified target directory (`/home/vz/gen/`).

The code generator will generate:

Description	Filename
A configuration file	ConfigurationApp.nc
A nesC makefile	Makefile
A header file mapping colour sets	global.h
A set of components	e.g, DISDIO, DAO
A set of corresponding interfaces	e.g, DISDIOC, DAOC