*The man of science has learned to believe in justification, not by faith, but by verification.*

– Thomas Henry Huxley, 1866.

**University of Alberta**

# Behavioral Verification of Small Networks of State-Machines Built with Arduino-like Processors

by

**Parisa Delfani**

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

**Master of Science**

Department of Computing Science

*To my parents*
*For their unreserved love and unfailing support*

# Abstract

Inexpensive yet versatile limited-capability processors enable computing to be embedded in many kinds of devices and situations. Most applications are simple purpose-programmed reactive systems that interact with the environment through sensors and actuators. Because the processors are limited state-machines, and in principle can be fully specified, they are amenable to rigorous formal verification. Low cost wired and wireless connection schemes permit the easy aggregation of these processors into networks with both static and dynamic topologies. The resulting networks will often have unexpected behavior or emergent properties. This thesis is a step towards formally reasoning about such networks. Our contribution is a simple domain-specific programming environment that generates both the model for performing verification via model checking, and extracts executable code that runs on the Arduino computing platform.

# Acknowledgements

I would like to give my sincere thanks to my supervisor, Professor H. James Hoover, for his invaluable guidance and cooperation. This thesis would not have been possible without his help, support and patience. I would also like to thank Professor Ioanis Nikolaidis, whose remarkable attention to detail has greatly improved the quality of this thesis. I would like to express my appreciation to Professor Abram Hindle for letting us use his machine for running our experiments during this project. I extend special thanks to Morteza Ghandehari whose support and encouragement was always with me.

# Table of Contents

# List of Tables

# List of Figures

# List of Listings

# Chapter 1

# Introduction and Overview

The availability of low-cost versatile processors has enabled the embedding of small amounts of computing power into virtually every device. Although these processors are limited in terms of the traditional measures of CPU speed and memory size, they have a powerful capability to interact with the outside world through sensors and actuators. As a result of their flexibility, small computational devices have replaced electro-mechanical constructions in many products.

These processors also tend to perform specific tasks as part of an overall reactive system, taking input events from the environment and generating output events to the environment. As a result, they tend to be simple state-machines, although not necessarily programmed that way. The processors are limited in the amount of state they can maintain (often under 8 kB of memory), and the speed at which they can react to the environment (16 MHz clock, with 8-bit computations). Thus they are amenable to current model-checking techniques.

It is relatively simple to network these processors using a variety of methods, ranging from simple bit passing over digital input/outputs, to serial protocols over wires, to packets over wireless. The networks can have a static or a dynamic topology, depending on the application. Network links and individual processors can fail, and possibly recover. Thus collections of simple processors can be aggregated to form more complex computations, possibly with unexpected emergent behavior, that is difficult to reason about. This is an obvious situation in which to try verification. Thus this thesis asks:

> *Can the verification that is possible for single processors be scaled up to include small networks of such processors?*

For verification to work it must be possible to formalize the specifications of the system. But most problems are not well understood initially, and specifications often need to be co-developed with the creation of executable prototypes that enable developers and users to try the system out to see if it does what they intend. That is, systems for verification go hand-in-hand with systems for creating code. Thus we also ask:

> *What development framework enables both code generation and code verification?*

*How well do the code and verification model match?*

We were originally motivated by a bigger question: *How can we specify and build trustworthy sensor networks?* The PicOS operating system [9] used in sensor networks at the University of Alberta is based on a state and events model for processes, thus is has a natural fit into the model-checking tools for specification and verification. Although not pure state machines, is might be possible to do a simple transformation of a PicOS-based system into a model to be verified. In addition, PicOS is small and light-weight, easily comprehensible by one person, thus possible to model. We began our work by exploring if it was feasible to model and reason about PicOS and processes running under it. This was unsuccessful, as the state-space of a naive PicOS model is too big. But performing the modeling process gave us insight into potential simplifications that preserved much of the PicOS functionality.

When even PicOS proved to be too big to model in full detail, we decided to abstract even further, resulting in a simpler computation model that was feasible to encode directly using the NuSMV model-checking tools. This model was also directly implementable on Arduino micro-controllers, thus providing us with a live test-bed.

Because the translation into both NuSMV and executable code is straight-forward, this led to us developing a domain-specific language (DSL) which can be translated into a model to verify or into code to run. We call this combination of DSL, model-checking, and executable code by the name *ArduinOS* .

When then validated the ArduinOS approach by specifying, developing, and verifying a sequence of increasingly complex programs: blinking light, token passing, ring-orientation, and cluster head.

## 1.1 Structure and Main Results

The structure and main results of this thesis are as follows:

**Chapter 2** (***Background***) presents the research background, explaining the relevant research areas of the problem discussed in this thesis, and focuses on the power points and pitfalls of each research, and how this thesis can contribute to the field.

**Chapter 3** (***PicOS Feasibility Study***) describes in detail our initial modeling study of PicOS in NuSMV in an attempt to answer the question whether PicOS is a small enough operating system and application environment to model. It points out the general characteristics of PicOS application and NuSMV model checker, and reveals the complexities of formal verification of even a simple PicOS application.

Our study on model checking of PicOS (explained in Chapter 3) reveals that the structure of PicOS applications makes them appropriate and easy to be modeled to some extent. PicOS applications are a combination of finite state machines (FSMs) defined as which can be activated one at a time on a node. Therefore, each processor node actually acts as a multi threaded component where threads are individual FSMs. The operating system underlying PicOS applications follows a clear event driven strategy for coordinating the program execution and scheduling of FSMs. Using these event based and behavioral notions for designing applications, PicOS becomes capable of covering wide range of algorithm, interesting to model check. In addition, its state based construction has the same structure as model-checking languages like NuSMV, which makes the modeling procedure easier.

The disappointment of the feasibility study was that we cannot model-check real PicOS applications — even the simplest PicOS systems are too large. Many factors contribute to the large model. For example, the choice of event dispatch and process scheduling algorithm will affect how many alternative paths exist in the model. For instance, if a set of events trigger at the same time, PicOS may give them the same priority. If a process is waiting for one of them it can select one of the events randomly, or it may select the one that its "wait for event" command came first. Another complexity we face is that PicOS uses wireless communication, so has the real-world issues of packet losses, conflicts in receiving packets at the same time, and the requirement for policies for sending big packets containing sender identifiers and related values. This alone creates a big complex model.

As a result, a fully faithful model of PicOS, even on a machine with 64 GB of memory, was impossible.

**Chapter 4** (***Framework***) presents our model-driven approach for analysis, design, and implementation of our verification framework. It explains why and how the essence of PicOS was distilled into an even simpler Arduino-based OS (ArduinOS), and how a restricted language and computational model (DSL) is applied as the source of ArduinOS tractably verifiable application. We made the following key assumptions:

1. All processes would be pure finite-state machines.
2. Within a processor, processes communicate via events.
3. Processes are responsible for coordinating their scheduling, there is no operating system involved.
4. Processors are asynchronous, and can run at different speeds.
5. Processors communicate over a network composed of binary channels consisting of lines (continuous channels) between digital input and output.

It is possible to model this simplified world directly using the NuSMV model-checking tools.

Under these assumptions we created a new programming environment in which the Arduino platform replaces the programmable sensor nodes of PicOS. This simpler Arduino-based OS (ArduinOS) eliminates the complicated packet sending protocols and replaces them with simple communication channels.

We also further simplified the PicOS FSM programming model, creating a domain specific language (DSL) and computational model that covers many of the same state-based process definitions and event driven coordinating strategies in lower layers. This DSL is translatable to both Arduino programs and their corresponding model for formal verification.

**Chapter 5** (***Experiments***) presents a sequence of increasingly complex case studies of ArduinOS. The case studies are mainly focused on the basic features covered by the modeling framework in order to reveal the accuracy of framework in modeling the sensor networks fundamentals.

**Chapter 6** (***Conclusion and Future Work***) presents a summary of the thesis and the results. The degree to which the primary objective was achieved, and the shortcomings of the thesis are discussed. Finally, suggestions for further research are made.

# Chapter 2

# Background and Related Work

This thesis combines three major areas; behavioral event based programming, formal verification, and sensor network applications. Each of these fields has specific features that make it challenging to study. However, our combination does not involve any deep understanding of each area, and we assume that the reader has a passing familiarity with all three areas.

In this chapter, we provide a deeper introduction to the methods we used in the remainder of this thesis, and then provide a brief review of some studies performed in the context of behavioral programming and sensor networks and their formal verifications. Note that sensor networks are only a motivation, we will use them as a rich source of examples; therefore, we will not include a comprehensive survey in that context.

## 2.1 Background

### 2.1.1 Formal Verification Methods

The process of building high quality systems involves two main styles of question:

*Validation - did we build the right system?*

*Verification - did we build the system right?*

Getting the specifications right for a system is in many cases as difficult as determining that the system is correctly implemented. Furthermore, specifications and implementation interact. A small change in one can make a big difference in the complexity of the other. So one can never expect to fully specify a system prior to actually building it.

Thus one of the goals of applied formal methods is to integrate the activities of specification, implementation, testing, and verification so that information can be mechanically exchanged among them. Automating the exchange removes a major source of error caused by hand crafted translation.

The motivation behind our work was integrating sensor network implementation with verification so that the implementation meets the specification. But which verification approach to apply?

- Theorem-proving. The relationship between a specification and an implementation is regarded as a theorem in logic, to be proved within the context of a proof calculus, where the implementation provides axioms and assumptions that the proof can draw upon.

- Model checking. The specification is in the form of a logic formula, the truth of which is determined with respect to a semantic model provided by an implementation.

- Equivalence checking. The equivalence of a specification and an implementation is checked, e.g. equivalence of functions, equivalence of finite-state automata, etc.

- Language containment. The language representing an implementation is shown to be contained in the language representing a specification.

In general, is it easier to perform specification, implementation, and verification if they all have the same conceptual model. For example, pure functional programs lend themselves to conventional equational-style proofs; finite-state-machines are conceptually matched to model checking. Because PicOS uses a FSM style for implementation, it was clear that we should start with model-checking as our verification technique.

### 2.1.2 Model Checking

According to the definition [5, 12], "Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model.

A model checker takes a model and some property specifications, and then explores all possible execution paths of the system in a relatively efficient manner to determine whether the properties are satisfied along each path. If a property is satisfied in all the execution paths the model checker will confirm that the property holds. If not, the model checker will provide (where possible) a counter example execution path showing how the system could reach a state that violates the property. In the case of a concrete counter example, the user can simulate the path shown by the counterexample and obtain useful debugging information [5].

The process of system verification using model checking consists of the following phases, which are performed numerous times as the model is refined:

1. Modeling: the system being implemented is modeled as a finite-state machine using, in our case, the NuSMV model description language. Since this is often an error-prone step if done by hand, the system model should be validated to check if it is the one desired by checking some observable and expected properties. This is true even if the model is extracted from another source, like code.

2. Specifying : formalize the properties to be verified using, in our case, the LTL, CTL property specification language.

3. Checking: run the model checker to check whether the property specifications are satisfied by the system model. The model checker indicates if the property is true for all possible states, or it is violated by some execution path (the counterexample).

4. Analysis: deal with the outcomes of the model checker. There are three possible outcomes from the checking phase:

   - If a given property is satisfied, it is important to verify that it is not vacuously true.
   - If a property is violated and a counterexample is provided by model checker, the user should analyze the counter example by simulating (if possible) to determine if the model or the property is at fault.
   - The model checking does not terminate successfully due to lack of memory or time. In this case a major revue of the model is necessary in order to reduce the size of the explored state space by reducing the number of bits of state in the model, or by restricting the possible transitions in the model.

### 2.1.3 A Brief Review Of Temporal Logic

Usually in the normal operational semantics of machines, we talk about explicit sequences of states, indexed by time. As a system runs, it passes through a sequence of states, where the next state $S(t+1)$ at time $t+1$ depends on the current state $S(t)$. This means that any assertion about the state of the system also has to mention the time at which this assertion is being made.

The purpose of a temporal logic is to enable one to talk about the behavior of a system over time without having to explicitly mention the time variable. The temporal logic lets you push the indexing details behind the scenes, but at some loss of expressive power. The two most common temporal logics used for finite state systems are CTL and LTL.

#### CTL - Computation Tree Logic

CTL is a temporal logic that lets one talk about the future. Furthermore, it is a branching time logic that lets one talk about possible futures.

CTL formulas consist of the usual atomic propositional logic formulas, plus temporal connectives. The propositional logic formulas are expressions about the state of the system. The temporal connectives are expressions about paths into the future that the state of the system can follow.

Temporal connectives are pairs of symbols. They talk about what can happen from the current state. The "current" state is the one being described in the formula. The future is infinite, i.e. the computation doesn't halt, although it can stay in the same state forever. The first member of the pair constituting the temporal connective is one of the letters:

**A** - meaning on all paths from the "current" state, read as "inevitably"

**B** - meaning on at least one path from the “current” state, read as “possibly”

The second member of the pair is one of the followings:

**X** - meaning the next state

**G** - meaning all future states, read as “globally”

**F** - meaning some future state

**U** - meaning until

The system state $S$ is described by a collection of state variables, each variable with a finite collection of values. Suppose that the system is in some state $S$. *The future of* $S$ is all the possible paths that the system can follow beginning with $S$. The future of $S$, by definition, includes $S$ (0 time into the future).

The temporal connectives are described below, using the input syntax of the NuSMV model checker. Let $\phi$ be a CTL formula. The simplest such formula is just a propositional formula involving values of the state variables.

$\phi$ is TRUE iff it is satisfied by the current state $S$.

AX ( $\phi$ ) is TRUE iff $\phi$ is TRUE for every immediate successor to state $S$.

AG ( $\phi$ ) is TRUE iff $\phi$ is TRUE for every successor to state $S$, including $S$. That is, $\phi$ is TRUE for all states on all paths into the future from $S$.

AF ( $\phi$ ) is TRUE iff on all paths into the future from $S$, there is a state where $\phi$ holds.

A [ $\phi$ U $\theta$ ] is TRUE iff all paths starting in state $S$ satisfy $\phi$ until the reach a state in which $\theta$ holds.

EX ( $\phi$ ) is TRUE iff $\phi$ is TRUE for at least one immediate successor to state $S$.

EG ( $\phi$ ) if TRUE iff there is a path from $S$ into the future for which $\phi$ holds for every state on the path, including $S$.

EF ( $\phi$ ) is TRUE iff there exists a path into the future from $S$ on which there is a state where $\phi$ holds.

E [ $\phi$ U $\theta$ ] is TRUE iff there exists a path starting in state $S$ that satisfies $\phi$ until reaching a state in which $\theta$ holds.

The following equivalences hold for CTL formulas:

$$
\begin{gathered}
!\ \mathrm{AF}\ \phi \iff \mathrm{EG}\ !\ \phi \\
!\ \mathrm{EF}\ \phi \iff \mathrm{AG}\ !\ \phi \\
!\ \mathrm{AX}\ \phi \iff \mathrm{EX}\ !\ \phi \\
\phi \iff \mathrm{A}\ [\ \mathrm{TRUE\ U}\ \phi\ ] \\
\mathrm{EF}\ \phi \iff \mathrm{E}\ [\ \mathrm{TRUE\ U}\ \phi\ ] \\
\mathrm{AG}\ \phi \iff \phi\ \&\ \mathrm{AX\ AG}\ \phi \\
\mathrm{EG}\ \phi \iff \phi\ \&\ \mathrm{EX\ EG}\ \phi \\
\mathrm{AF}\ \phi \iff \phi\ |\ \mathrm{AX\ AF}\ \phi \\
\mathrm{EF}\ \phi \iff \phi\ |\ \mathrm{EX\ EF}\ \phi \\
\mathrm{A}[\ \phi\ \mathrm{U}\ \theta\ ] \iff \theta\ |\ (\phi\ \&\ \mathrm{AX\ A}[\phi\ \mathrm{U}\ \theta\ ]\ ) \\
\mathrm{E}[\ \phi\ \mathrm{U}\ \theta\ ] \iff \theta\ |\ (\phi\ \&\ \mathrm{EX\ E}[\phi\ \mathrm{U}\ \theta\ ]\ ) \\
\mathrm{A}\ [\ \phi\ \mathrm{U}\ \theta\ ] \iff\ !\ (\ \mathrm{E}[\ !\theta\ \mathrm{U}\ (\ !\phi\ \&\ !\theta\ )\ ]\ |\ \mathrm{EG}\ !\theta\ )
\end{gathered}
$$

Here is an example of things we might want to say about Mutual Exclusion using CTL. Suppose we are talking about two processes P1, P2 that share data. Then we would like to ensure the following kinds of properties for our mutual exclusion protocol:

*Safety:* the protocol allows only one process to be in its critical section at any time.

AG ! ( Critical[P1] & [P2] )

*Liveness:* whenever any process wants to enter its critical section, it will eventually be permitted to do so.

AG ( Entering[P1] -> AF Critical[P1] ) & ( Entering[P2] -> AF Critical[P2] )

*Non-blocking:* a process can always request to enter its critical section.

AG ( Idle[P1] -> EX Entering[P1] ) & ( Idle[P2] -> EX Entering[P2] )

*No strict sequencing:* processes need not enter their critical section in strict sequence.

EF ( Critical[P1] & E[ Critical[P1] U ( !Critical[P1] & E[ !Critical[P2] U Critical[P1] ]
) ] )

Note: Critical, Entering, Idle look like predicates, but are just shorthand for a propositional function of the current state that reflects this situation.

**LTL - Linear Temporal Logic**

LTL is a temporal logic that lets us talk about the future and the past. It is linear, in that it lets us talk about each possible path into the future, but without considering branching. That is we cannot talk about branching within the context of a specification - we have only the path we are following.

Similar to CTL, LTL formulas consist of the usual atomic propositional logic formulas, plus temporal connectives. The propositional logic formulas are expressions about the state of the system. The temporal connectives are expressions about paths into the future that the state of the system is following.

LTL is CTL without the A and E connectives, except that you assume an A (all paths) connective in front of the LTL specification. LTL connectives talk about what can happen from the current state. The "current" state is the one being described in the formula. The future is a single infinite path without branching. As for CTL the computation does not halt, although it can stay in the same state forever.

The LTL connectives are

X - meaning the next state

G - meaning all future states, read as "globally"

F - meaning some future state

U - meaning until

The temporal connectives are described below. Suppose that the system is in some state $S$. The future of $S$, by definition, includes $S$. Consider all the possible paths starting in $S$.

$\phi$ is TRUE iff it is satisfied by the current state $S$.

X ( $\phi$ ) is TRUE iff for all paths from $S$ $\phi$ holds for the immediate successor to state $S$.

G ( $\phi$ ) is TRUE iff for all paths from $S$ $\phi$ holds on all states on the path.

F ( $\phi$ ) is TRUE iff for all paths from $S$, there is a state on the path where $\phi$ holds.

($\phi$ U $\theta$ ) is TRUE iff for all paths from $S$ $\phi$ holds until state occurs in which theta holds.

CTL and LTL are not equivalent. There are things we might want to say using in one that we cannot say in the other. For example

FG(p) - along every path from initial state $S$ there is a state from which p will hold forever. This cannot be expressed in CTL.

AG(EF p) - for every path from the initial state there exists at least one future state in which p holds. This cannot be expressed in LTL.

### 2.1.4 Using NuSMV To Model Randomness

The behavior of a FSM is driven by its current state and the possible events that can arrive and be recognized by that state. The system model indicates these situations by using non-deterministic assignment to generate the forks in the execution path. All of these potential paths are explored, with no memory of any previous states.

This poses a problem for systems where random choices are made. Many systems require random coin tosses in order to break symmetries that would cause the system to lock up and not make progress. For example, collision resolution in an Ethernet, or next-actor selection in a token passing protocol.

But a random event cannot be modeled by a non-deterministic choice because the random event is part of a sequence of events that must be consistent with a specific probability distribution. That distribution will in part dictate the expected behavior of the symmetry-breaking process. For example, when using two fair coins to elect a leader in a pair, the expected number of simultaneous tosses before the two coins differ is 2. The model checker knows nothing about the expected distribution, and so will explore the infinite path on which the two coins always come up the same. But the probability of this path is zero, so it should be removed from the set of explored paths. So then the question becomes how large a finite prefix of this path should be explored in the checking process?

There is no single answer to this. The goal is to track the past for enough transitions to be able to detect a failure to break symmetry. Then to use that information to prevent any further exploration along that path.

Here is an example of modeling randomness in an implementation of the Hoover-Rudnicki token passing protocol [20]. Since the expected length of path to break symmetry is 1.5 steps[1] we only need to keep a history of 2 sets of random choices. If the histories of each node match, then we use a TRANS constraint in NuSMV to prevent any further exploration along that path.

```
-- implementation of the Hoover-Rudnicki token passing protocol
-- between two nodes.

MODULE main
VAR
    N0 : node(N1.mode);
    N1 : node(N0.mode);

-- To model random tosses, we need to rule out sequences of tosses
-- that have expected value of 0.  With a 3-sided coin, the
   expected
-- number of pairs of tosses before  they differ is less than 2,
   so
-- we only need to keep a history of two tosses in order to model
   the
-- use of randomness to break symmetry.  this specification rules
   out
-- all transition historys in which the nodes make identical
   tosses.

TRANS next(N0.history) != next(N1.history);

-- this is the property we want to verify:
-- synchronization always occurs at some point in the future, and
-- once synchronized, they follow this sequence

SPEC AF ( N0.history = N1.history | (
    A [ N0.mode = W & N1.mode = R U (
```

[1]Since the probability of not being different is $1/3$, the expected path length to being different is $3/2 = \sum_{n=1}^{\infty} 2n/3^n$

```
    A [ N0.mode = W & N1.mode = P U (
    A [ N0.mode = R & N1.mode = P U (
    A [ N0.mode = R & N1.mode = W U (
    A [ N0.mode = P & N1.mode = W U (
    A [ N0.mode = P & N1.mode = R U (
    N0.mode = W & N1.mode = R ) ]
        ) ]
        ) ]
        ) ]
        ) ]
        ) ]
        ) )

MODULE node(in_mode)
VAR
    -- the mode of the node, as in the protocol
    mode: {R, W, P};

    -- the coin toss history, 2-bits per toss, oldest on left,
    -- 00 - undefined
    -- 01 - W
    -- 10 - P
    -- 11 - R
    history : word[4];
ASSIGN
    init(mode) := {W, P, R};
    init(history) := 0b4_0000;

    next(mode) := case
        mode = in_mode: {W, P, R};

        mode = W & in_mode = P: R;
        mode = W & in_mode = R: W;
        mode = P & in_mode = W: P;
        mode = P & in_mode = R: W;
        mode = R & in_mode = W: P;
        mode = R & in_mode = P: R;
        esac;

    -- record the toss just made
    next(history) := case
        next(mode) = W : history << 2 | 0b4_01;
        next(mode) = P : history << 2 | 0b4_10;
        next(mode) = R : history << 2 | 0b4_11;
        esac;
```

Listing 2.1: Token Passing With Randomness

### 2.1.5 Using NuSMV To Model Process Scheduling

When concurrent processes are present in a system, there is always the question of how they will be scheduled to interact. When exploring the correctness of a concurrent algorithm, the scheduling discipline must be considered. Do all processes in the model make a transition at each step? Do processes on different processors operate synchronously? Does the scheduler have to be fair? Is it allowed to starve a process? Although processes and a variety of scheduling concerns are built into NuSMV, one has to consider the scheduling that will actually occur in our particular systems. An algorithm that is correct under one scheduling discipline may not be correct under another.

Here is an example of how we add scheduling to the token passing algorithm just described. The scheduler ensures that either both processes run simultaneously, or if one process runs alone, then the next time the other process or both will run.

```
-- implementation of the Hoover-Rudnicki token passing protocol
-- between two nodes, with explicit scheduling
MODULE main
VAR
    -- when true process 0, 1 will make one transition
    run_0: boolean;
    run_1: boolean;

    N0 : node(run_0, N1.mode);
    N1 : node(run_1, N0.mode);

    -- schedule node 0, 1, or both to execute next
    schedule : {S0, S1, S01};

    -- schedule history, previous step
    shistory : {S0, S1, S01};

ASSIGN
    init(schedule) := S01;
    init(shistory) := S01;
    init(run_0) := TRUE;
    init(run_1) := TRUE;

    -- initialize nodes in arbitrary states
    init(N0.mode) := {R, W, P};
    init(N1.mode) := {R, W, P};

    -- or start them in the same state and see if they converge
    -- init(N0.mode) := R;
    -- init(N1.mode) := R;

    -- record current schedule
    next(shistory) := schedule;

    -- a bad schedule, allows starvation
    -- next(schedule) := {S0, S1, S01};

    -- a synchronous schedule, both processes run every step
    -- next(schedule) := S01;

    -- a round-robin-ish schedule
    next(schedule) := case
        shistory = S0 : {S1, S01};
        shistory = S1 : {S0, S01};
        shistory = S01 : {S0, S1};
        esac;

    -- run the appropriate process
    next(run_0) :=
        (next(schedule) = S0) | (next(schedule) = S01);

    next(run_1) :=
        (next(schedule) = S1) | (next(schedule) = S01);

-- this forces the coin toss history to be different
-- this will break symmetry
TRANS next(N0.history) != next(N1.history);

-- synchronization always occurs at some point in the future, and
-- once synchronized, they follow this sequence
SPEC AF (
    A [ N0.mode = W & N1.mode = R U (
    A [ N0.mode = W & N1.mode = P U (
    A [ N0.mode = R & N1.mode = P U (
    A [ N0.mode = R & N1.mode = W U (
    A [ N0.mode = P & N1.mode = W U (
    A [ N0.mode = P & N1.mode = R U (
    N0.mode = W & N1.mode = R ) ]
        ) ]
        ) ]
        ) ]
        ) ]
        ) ]
        )

MODULE node(run, in_mode)
VAR
    mode: {R, W, P};
    -- length depends on expected number of tosses
    history : word[4];
ASSIGN
    init(history) := 0b4_0000;

    next(mode) := case
        -- transition if running
        run : case
            mode = in_mode: {W, P, R};

            mode = W & in_mode = P: R;
            mode = W & in_mode = R: W;
            mode = P & in_mode = W: P;
            mode = P & in_mode = R: W;
            mode = R & in_mode = W: P;
            mode = R & in_mode = P: R;
            esac;

        -- otherwise no change in state
        TRUE: mode;
        esac;

    next(history) := case
        -- history update if running
        run : case
            next(mode) = W : history << 2 | 0b4_01;
            next(mode) = P : history << 2 | 0b4_10;
            next(mode) = R : history << 2 | 0b4_11;
            esac;

        -- otherwise no change in history
        TRUE: history;
        esac;
```
Listing 2.2: Token Passing With Scheduling

## 2.2 Related Work

Research with related goals to ours can be categorize in two main groups. The first group contains the studies that focus on behavioral programming frameworks that are verifiable by formal methods and applicable for event driven applications. The second group of studies contains the research on verification of specific languages or infrastructures for small reactive systems or sensor network applications.

### 2.2.1 Behavioral Event Collaborative State Based Programming

Behavioral programming [18] is a general framework for developing applications consisting of independent components working together to provide a behavior.

- Each individual component is specified in terms of its individual behavior under different scenarios for interacting with the world through events. In principal, this individual behavior is simple enough that it should be easily verifiable.
- The components collaborate with each other by exchanging messages over a network. The possible connections between components are established using a publish/subscribe protocol, in which the publisher is the node that sends out the event and subscribers are the nodes waiting to react to that specific event. The result is a network of communicating processors that has some desired behavior. Because it involves concurrent processes and processors, this resulting behavior is more difficult to specify, and unexpected behavior can emerge.

The activities of specification, verification, implementation, and testing all interact. In particular, the language chosen for one of these can seriously impact the feasibility of the other. For example, implementation in a general procedural language (like C) can make it very difficult to specify the desired behavior of the system. As a general rule, the most important task is matching the programming language to the verification tools. For a broad selection of computations, especially reactive systems involving embedded computation, the most suitable computation model and programming language is that of finite-state machines (FSMs). FSMs are easy to specify, and can naturally be translated into implementation code. Extracting models for verification is similarly simple, although there are potential difficulties with scheduling processes and using random numbers.

Where FSMs are not suitable is where there is additional state that cannot easily be abstracted away — for example, the workflow for processing an order is a simple FSM, while maintaining a list of items on an order and computing their total is not.

The PicOS sensor network environment uses this philosophy, but in addition augments the state machines with additional global variables (of small size). PicOS provides an execution context (thread) for each FSM, an event dispatching mechanism, and a scheduler to decide which process makes its next transition. Thus PicOS is a more complex layer above the simple behavioral model.

### 2.2.2 Formal Behavioral Programming

Emphasizing the behavioral and scenario-based requirement analysis and system specifications, Harel et al. proposed the behavioral programming framework to facilitate software development [17, 18], specially scenario-based development of reactive systems. They introduced behavioral application for software consisting of independent constructs called behavior threads (b-threads) each of which describes a scenario. In other words, they defined b-threads as the backbones of the software to be developed. Each b-thread should be designed for controlling and coordinating a specific behavior of the system, and it may involve a set of objects or system components. Therefore, the software is a combination of b-threads.

B-threads of a system run in parallel to each other, and get synced in synchronization points, i.e., each b-thread runs their normal flow until they requires synchronization, then the thread waits until all other running b-threads reach synchronization points in their own flow. At synchronization points an enhanced publisher/subscriber pattern is set up. Each b-thread specifies three sets of events: 1) requested events that are the ones requested to be triggered (i.e., the b-thread wants the event to be triggered, so one of the requested events may be triggered in the next step); 2) wait-for events that are not requested to be triggered but can notify the b-thread if occurred; 3) blocked events that are forbidden to be triggered. After all the b-threads reach their synchronization point and present their requested, wait-for and blocked events, an event which is requested by at least of the b-threads but is not blocked by any of them will be chosen (perhaps arbitrarily) and notifies all threads requesting or waiting for it. The notified b-threads then start running their normal flow until they reach another synchronization point and present a new sets of events as their requested, wait-for or blocked ones. Accordingly, b-threads coordinate the behavior of the software by generating a flow of events and being notified by triggered events.

Harel et al. also provided a Java library called BPJ which consists the implementation for the protocol idioms and coordination mechanism of b-thread execution and synchronization. They follow a bottom-up approach in providing scenario based programming, as they started from a general-purpose language (Java) and then added programming tools (the BPJ library) to code behavioral elements of the system. Therefore, they aimed at providing a programming context for facilitating behavioral and scenario-based programming. Their study was based on Java platform and JVM thread execution, which limits its extensibility to sensor network applications that use a simpler operating system and coordinating mechanisms than JVM. However, the notion of behavioral programming was proposed for software development in general, and BPJ was just a demonstration of its viability.

The event-based coordination of modules in behavioral programming makes it ideal for being used on event collaborative applications in small reactive systems, e.g. applications for each node of a sensor network. For instance Shimony et al. used the same coordination approach as b-thread coordination to extend PicOS and present a new set of coordination constructs aiming at improving the high-level characteristics of PicOS WSN programs [24].

Beside the applicability of the behavioral approach in reactive systems, the verification and model checking of the scenario based applications developed by this method is an interesting field of research, and also makes it interesting for our thesis approach. Following the proposal of behavioral programming in [17], Harel et al. provided a methodology for model checking-assisted development of behavioral programs; they also presented a model checker (BPmc) for supporting the methodology and verification of behavioral programs written in Java using BPJ [16]. BPmc is actually embedded in BPJ, and therefore bypasses the need for transforming the system implementation to a model understandable by the model checker.

The algorithm used by BPmc for model checking executes the participating b-threads and explores all the state space to find the state in which the property is violated. The tool keeps track of the visited states, using Apache Javaflow package. Whenever the tool reaches a previously visited state it backtracks the execution path, by restoring the state saved by Javaflow, and tries to explore other states. Anytime the tool wants to go forward in an execution path and proceed to the next state, it selects an event to trigger, and that may result to notifying some b-threads and branching an unvisited state space. Using this strategy, BPmc explores the states using breadth-first or depth-first searches.

Addressing the viability of using model checking techniques for having trustworthy behavioral reactive software, our thesis pursues the same goals as verifiable behavioral programming framework. However, we focus on employing the proposed event-based coordinating approach specifically for reactive systems, and design software executable on sensor network-like devices. Accordingly, our approach will not suffer from the limitations of the research provided in [16] that is limited by Java-based assumptions, such as Java thread scheduling, thread data, etc. For instance, while the proposed BPmc tool is based on Java platform and follows its thread scheduling strategies, we provide a more flexible framework for programming these kernel layer functionalities and provide the potential of scheduling alternatives based on the needed underlying logics. Besides, not depending on Java based assumptions; we provide the possibility of data sharing between behavioral threads.

Another notable study that emphasizes the system behavior, from atomic constructs to the composition of components, is the research done by Bliudze et al [7, 8]. They proposed the BIP language (behavior, interaction, priority) as a framework for designing component based systems [7]. In their viewpoint, components of a system can be specified by their behavior, and are combined and communicating based on their connectors and priority rules. In BIP language, the backbones of any system structure consists of 1) atomic components, a set of components of which behavior has been specified as a set of transitions; 2) connectors, which are used for specifying interactions between atomic components; 3) priority rules, which are used to coordinate the interaction based on the state of the integrated atomic components, and restrict non-determinism of the interactions. Accordingly, BIP allows construction of complex structures from atomic components, and also allows using strongly synchronized interactions between components.

BIP focuses on creating a system that is correct-by-construction, and satisfying the expected behavioral requirements of the system. Addressing this in the context of component based systems, Sifakis et al. provided a formalized semantic for assembling components by glue operators, and proposed a expressiveness comparison for it [8]. In their studies, they considered a composition of components, where each component can be specified by its behavior (its states and transitions), and the components are represented in some semantic domains. Glue refers to any BPI operator (connectors and rules) and should only restrict the behavior of its arguments (the atomic components) without adding new one. Sifakis et al. proposed a definition of glue where operators are characterized as sets of rules specifying the transition relation of composite components from the transition relations of their constituents [8]. Therefore they consider glue operators only as behavior transformers and examine their role in composition of components and building a system that is correct by construction.

A closer look at the BIP language reveals that its notion for constructing heterogeneous systems is the same as our approach toward structuring applications. For instance, assuming the atomic components of BIP language as individual nodes of the network or the executable threads in a node, their connectors are actually the channels through which the nodes/processes can communicate and send or receive events, and the priority rules are the coordination rules for handling the triggered events. Therefore, creating the behavior-based components is the goal in both studies.

We also follow the notion of “correct-by-construction” presented in their studies, but with a structural difference: they prove the correctness of component compositions assuming a single component has obtained the correct behavior and the components are glued properly. Whereas in our thesis, we assume that the lower level infrastructure layers, such as hardware and the event and signaling protocols, work correctly (the hardware verification of the sensor nodes is not the concern of this thesis). Then we examine the behavior of every single node (and their inner threads) in a layer on the top of the hardware layer, and further we prove the correctness of the network protocols and communications between nodes as another layer on top of the single node one. Accordingly, we model and verify our applications layer by layer.

### 2.2.3 Verification Of Sensor Networks

Sensor networks and the trustworthiness of their applications are the main motivation of this thesis, however, we will only use them as a rich source of examples and not necessarily the final environment for applying our approach.

Sensor networks consist of individual nodes that work independently and can communicate so that the network as a whole achieves some intended goal. Sensor network nodes are small in size and power: a small microprocessor running at a few MHz with at most 1 MB of memory. This limited memory is shared between the deployed programs and the data required or obtained by the node; therefore, only simple programs can be run on the hardware. In essence, since the sensor network programs are relatively simple and the devices are restricted with respect to several points, applying formal verification methods is feasible for proving the correctness of their applications. Additional difficulty is added to the computation due to unreliable communication, and incorrectly implemented processors. Failures are the norm, not the exception, in sensor networks. Thus sensor networks are a rich source of interesting problems.

In the context of sensor networks, the use of formal methods is relatively sparse, and it is strongly competitive with simulation techniques. One reason is that there are no off-the-shelf verification tool that implements formal method verification for sensor network [26], whereas there are accepted simulation tools such as NS2 [1] or the TOSSIM emulator [22, 23] which are applicable in verification of some sensor network application in TinyOS [21] platform. Although these tools are useful in providing an observable sample execution of the application, they all suffer from the simulation limitations and cannot provide a proof of correctness for the application. On the other hand, although some studies have focused on applying formal methods for verifying sensor network applications, the variety of properties aimed to be verified has made the domain covered by these studies so broad and sparse, for instance some studies specifically aimed at verification of MAC layer protocols, such as [10], while others assumed the MAC layer works properly and aimed at other aspects of a network application. In the following we provide some examples of different approaches to this problem

Among studies aiming the formal verification methods for verifying sensor network applications, some provided formal specifications of applications in order to make them mathematically provable during design [15]. Our research is similar to the this group of studies in the sense that we also need to detect all verifiable and specifiable properties of the applications first, and specify our programs based on what is feasible to model. Accordingly, a model based approach is provided in which the model and its constraints in abstracting the properties control what can be included in the applications.

Another group of studies focused on specific sensor network application programming language, SystemC [2], and attempted to apply formal verification methods for verifying its applications. For instance, Cimatti et al. provided a model checking approach for verifying SystemC applications [11]. They provided a translation procedure to extract a sequential program out of a SystemC application and change it to corresponding models in which the threads are coordinated by a scheduler. This was similar to the approach we followed at the beginning of our thesis when trying to see if modeling PicOS applications are feasible or not. We modeled PicOS and orchestrated its processes by a simulated PicOS scheduler. However, the huge low level networking details were needed to be considered while modeling PicOS network applications which resulted to unrealistic complex models.

Another important field in the context of sensor network is that of probabilistic algorithms and their analysis. The behavior of sensor network is probabilistic since the nodes are randomly distributed in the environment, and they achieve probabilistic behavior due to the collision avoidance mechanism for the wireless communications and the memory saving processes [13]. For verifying such networks, the application of simulation techniques are inadequate since on the one hand the number of nodes in the network is extremely large, and on the other hand experiments have to be repeated sufficiently often to gain a considerable high precision [26]. Therefore, providing a thorough verification is infeasible in these networks.

Addressing this problem, Demaille et al. employed Approximate Probabilistic Model Checking (APMC) [19] in order to approximately compute the probability that a model verify a specification [13]. In their research they modeled simplified sensor network in which the goal is to detect the occurrence of an event on a grid of arbitrary size and to react to the event by sending a signal to some specific nodes. They also examined some properties on the network model using the approximate probabilistic model checking technique to see if the model satisfies the specifications such as the probability that the event occurred in a grid will be detected without error is greater than $0.99$.

In our thesis, we also assume the different branchings that may occur in the execution path of an application. However, if the branches result to an unfair randomness situation, we just ignore that unfair path and aim to verify the application in a fair environment. In other words, we aim to prove that the probability that the model verify the intended specification is 100 percent in fair environments.

# Chapter 3

# Feasibility Study - Modeling PicOS Applications In NuSMV

This chapter asks if the PicOS sensor-network operating system and applications are simple enough to be modeled and checked with the NuSMV model checker. We provide the fundamentals of our modeling approach and explain why we need to migrate to ArduinOS in order to make any progress (Chapter 4). The main take-away from the rather intense amount of detail in this chapter is that it is astonishingly difficult to model even a relatively simple system like a PicOS application.

This chapter is organized as follows. In Section 3.1 we explore our modeling source and destination environments, we introduce the characteristics of PicOS operating system, the state-based structure of its applications, and the properties interesting to be verified in this context. We also introduce NuSMV model checking framework as the reasoning system, and explain its limitations for modeling specific features. Furthermore, we include a comprehensive example of a PicOS application containing all semantics to be modeled in NuSMV. We present the state diagram and PicOS representation of the application and provide the corresponding NuSMV model as an example of the goal model to be achieved by the end of this chapter.

In Section 3.2 we analyze the PicOS properties and point out the visible and implicit requirements to be modeled, using two main examples. Accordingly, the modeling assumptions, specifying the fundamental properties and constraining the PicOS functionalities will be pointed out and create the basis of our modeling schema.

In Section 3.3 a high level logical architecture of the system is provided. In Section 3.4 the component-based design of the modeling schema and the modeling elements are introduced and their responsibilities in covering the fundamental PicOS logics are explained. Having the big picture of what the modeling approach is and its outcome, we specify the modeling procedure in more details in Appendix A. We present the process of extracting model out of a PicOS application through detailed design of each modeling element and its implementation in NuSMV. We provide the applicability of our modeling approach in validating the PicOS kernel model, and also verification of simple PicOS applications respectively in Sections 3.5 and 3.6.

Finally, we conclude our PicOS modeling experiment in Section 3.7 and discuss the reasons why PicOS applications are not small enough to be feasibly verified using our detailed approach of modeling the operating system.

After reading this chapter, you should understand the characteristics of PicOS operating system and applications, and see why PicOS is not a small enough sensor network framework to be model checked.

## 3.1 Context Identification

The model of an object can be affected by two factors: 1) the properties of the main object, 2) the constraints of the modeling environment. For us this means the properties and logic of PicOS applications, and the constraints that are forced by NuSMV. In this section we briefly introduce these factors.

### 3.1.1 PicOS

PicOS is a small operating system for organizing multiple activities of embedded reactive applications [3]. PicOS is written in C for a microcontroller with limited on-chip RAM (e.g., 4 kB). Although small, PicOS provides a flavor of multitasking and event-driven inter-process communication [4]. PicOS inherits its programming paradigm from SMURPH (a.k.a. SIDE) [14], which is a specification and simulation environment for low-level communication protocols and reactive systems [25]. Being related to SIDE, PicOS applications can be emulated in a SIDE-based realistic virtual environment called VUE2 (Virtual Underlay Emulation Engine), and the virtual behavior of the applications can be observed in this environment. In this section we focus on introducing how PicOS applications are developed and provide a general view of the task coordination and CPU allocation routines performed by the operating system while organizing the execution of the tasks [1].

PicOS applications consist of multiple nodes operating simultaneously. Each node is a single processor. A PicOS application on a node consists of some tasks, called processes, with at most one task active at a time. Each node has its own CPU scheduler that decides which task (a.k.a. process) should be performed next. In order to multitask, the node's CPU is multiplexed among the processes at the granularity of state-transitions. While the CPU is allocated to a process, the task owns the CPU and no other process can get the CPU until the owner releases the CPU.

Each process is described by its code and data in a FSM based style. The code is imperative-style C organized in a state-machine-based way, with additional per-process data composed of basic types such as Boolean and Integer. As a FSM, each process can have multiple defined entries. An entry is a state in the FSM and can contain rules for triggering events (to invoke other processes), waiting for some events to be triggered by other processes, setting the states in which they want to be woken up after an event is triggered, and releasing the CPU.

[1]For more details regarding PicOS operating system and applications refer to [3] and [4].

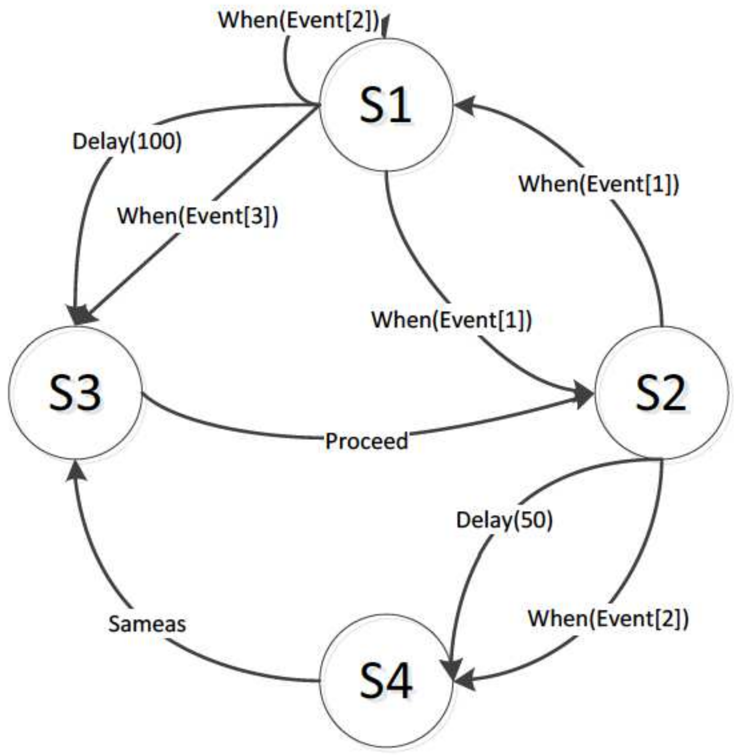

Figure 3.1: Comprehensive PicOS Command Example

In the context of PicOS applications each process can be in one of the three main states: waiting, ready (or runnable), and running. When a process is instantiated it goes to state ready, meaning it is waiting for its turn to get the CPU. When it gets the CPU, it transits to the running state (which is actually one of the defined entries in its PicOS FSM definition) and perform all the imperative statements coded for that state (changing the data, triggering events, jumping to other running entries, set events to wait for) and releases the CPU at the end. By releasing the CPU, the process goes to the waiting state (waiting for events). If any of the interesting events happens while the process is in waiting state, the kernel changes the process' state to ready (runnable) and the process becomes waiting to be scheduled for the CPU again.

A PicOS application is programmed by specifying an overall state-machine, and then writing the code to be run in each of the running states. In the state diagram representation of a PicOS application, we only specify running states (defined entries) as states and assume that wait and ready states are kernel level states which are not affecting the actual behavior of the system. The reason is that demonstrating all transitions to wait and ready states makes the state diagram less readable. Besides, what we actually are interested about the behavior of the application is resided in its running states.

Figure 3.1 and Listing 3.1 present the state machine of a single-processed application and its corresponding PicOS implementation respectively. This example demonstrates how a single-processed application is modeled in PicOS. The actual behavior of the node is actually programmed in its FSMs, except for root which is only responsible for instantiating the other FSMs in the application. This demonstrated application contains a fabricated process which has four states and contains various control statements, e.g. when, delay, proceed, and sameas. In fact, we made and used this process to illustrate the modeling of a relatively complex process.

```
#include "sysio.h"
#define EVENT1 (&e1)
#define EVENT2 (&e2)
#define EVENT3 (&e3)
byte e1, e2, e3;

Fsm simpleProcess() {
  entry S1:
    when(EVENT1, S2);
    when(EVENT2, S1);
    when(EVENT3, S3);
    delay(100, S3);
    release;

  entry S2:
    when(EVENT2, S4);
    when(EVENT1, S1);
    delay(50, S4);
    release;

  entry S3:
    proceed(S2);

  entry S4:
    sameas(S3);
}
fsm root() {
  entry ROOT_INIT:
    runfsm simpleProcess;
    release;
}
```

Listing 3.1: PicOS Code For Comprehensive Example

After a process performs all the programmed behaviors of its currently running entry, it can just *release* the CPU and go to sleep (i.e. wait state) either if it is waiting for some events or not, it can directly jump to another entry by using the *sameas* command (and move to a new entry without releasing the CPU), it can *proceed* to another entry (releasing the CPU and giving other processes the chance to run, but staying ready to run whenever scheduler lets it execute). Accordingly, wherever we have a transition labeled as *when(EVENT)* we expect the application to transmit to wait state, and stay there till it feels EVENT; then it transmits to ready state and stays there till the process is scheduled for allocating the CPU and going to the intended running state. Wherever *proceed* transition is used, the process releases the CPU but goes to ready state (instead of wait) and waits to be scheduled to run and transmit to the intended running state; therefore, it releases the CPU and gives the other ready processes the chance to get the CPU and progress. The *sameas* transition however makes the direct change from one running state to another one, without meeting wait or ready state.

Each node has a *root* FSM that starts automatically after reset, and is responsible for creating all other processes. Other FSMs can be instantiated by the root or other FSMs and put their Process Control Blocks (PCB) in the PCB Table (PCBT) of the node. New processes stay idle (in their ready state) until the CPU scheduler lets them execute. When a process releases the CPU, the scheduler walks the PCBT to find the first ready process and tells it to run. If the scheduler does not find any ready process (all processes are waiting for some events or interrupts), it keeps iterating over PCBT. Whenever a ready process is told to wake up, it knows in which entry it should wake up and run. The PCBT is the shared memory space between all processes, in which there is a structure for saving the information about each of the processes, such as the current state of the process, the event it is waiting for, the state in which the process should be woken up if an interesting event is triggered, etc. The main loop of a PicOS application, at the high level, is as follows:

1. The PicOS kernel assigns an identifier to each process at their first entry to the application, where they instantiated.
2. PCBT contains a root process.
3. At first, the kernel has the CPU, and schedules the first process to execute (root process).
4. Root runs and instantiates other processes, and their information will be added to PCBT as ready to run.
5. The kernel gets the CPU, notifies all processes about the triggered events and received interrupts form other nodes (if any) and changes the state of the ones waiting for those event/interrupts to ready. Then it goes over processes in PCBT, and schedules the first ready to run process it finds. If no ready process is in PCBT, it keeps iterating over PCBT.
6. The scheduled process gets the CPU, wakes up in the appropriate running state (start state at the first entry) and the specified state (if notified because of an event). It runs the code for the state transition, sets a list of interesting events to wait for, and finally releases the CPU.
7. (Repeat step 5-6)

Consequently, we can assume that the behavior of a node is the direct result of the current states of the processes. Therefore, how the processes transits from one state to another, and the sequence of these transitions will make the execution scenarios of the application. So, aiming at verifying the behavior of the application, all factors affecting these transitions should be considered. The way events and interrupts are triggered and handled by processes, the algorithm that the scheduler uses to iterate over PCBT and select the next process to run, and the way processes take and release CPU affect the behavior of the PicOS application and the way it should be modeled. Accordingly, all these parameters should be considered while modeling PicOS applications.

### 3.1.2 NuSMV

NuSMV is a symbolic model checker that allows for representation of synchronous and asynchronous finite state systems. It also allows for the analysis of specifications expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL).

In order to use NuSMV for modeling and verifying the systems, we need to model an abstraction of the systems. Then we should provide the CTL or LTL specifications that express what we expect the systems to do. By applying graph traversal techniques, NuSMV analyzes the specifications to see if the model can satisfy them or not. If a specification is false, NuSMV provides a counter example that shows an execution path in which the model cannot hold the specification.

The similar notions for finite state machines which exist in PicOS and NuSMV makes this model checker a good choice for achieving our goal. For example a FSM in PicOS can be assigned to a module in NuSMV. However, some NuSMV properties make modeling the applications (especially PicOS ones) more challenging. For instance, we cannot have any global variable in a NuSMV program. This means that a module cannot access any externally defined variable unless the symbol of the variable is passed to the module while instantiating it. Therefore, a module in NuSMV can only access the variables which are passed to it as input parameters or those which are defined in the body of the module. On the other hand, in a NuSMV model, any module which has access to the fully qualified name of a variable (and not just the module that has defined the variable) can assign values to the variable. However, every variable can be assigned just in a single module. This means that a variable is readable by anyone who can access it, but it is writable by just one of the modules that have access to it. Accordingly, even if we have the access to the variable and can read its value, we may not be able to assign values to it since the assign statement for that variable has been already written in another module. Moreover, this is not a syntactic issue, but it is a semantic one.

For example, assume that we write a setter module named setX which is responsible for assigning a new value to variable x. If we want to use this setter module in two sample modules, namely p0 and p1, which try to set x as 0 and 1 respectively, the code would be as shown in Listing 3.2. However, the assign statement is coded just in one module, since in runtime we have two instances of setX (one for p0 and one for p1), the model checker does not consider this code as valid since it would have to decide how to arbitrate inconsistent writes to a variable. Accordingly, assigning a value to a variable should be done in a singleton, that is, a single instance of a single module.

```
MODULE main
  VAR
    x : 0..10;
    p0 : sample(x,0);
    p1 : sample(x,1);

MODULE setX(x, newValue)
  ASSIGN
    next(x) := newValue

MODULE sample(x, value)
  VAR
    Setter : setX(x, value);
```

Listing 3.2: Non-Functional Shared Variable in NuSMV

Another limitation of NuSMV in modeling applications (especially PicOS applications) is that it doesn't support arrays of modules. This means that if we define PicOS FSMs as modules, we cannot have PCBT as an array of FSMs. Therefore, modeling PCBT as an iterable list of FSMs becomes more challenging.

Another feature of NuSMV that can affect our models is that all assign statements that are defined in its modules occur simultaneously in each clock. Dealing with this feature is challenging in modeling PicOS applications in which at most one process can be run in a node at a time. Moreover, all of the modifications should be managed to make sure the sequence of the tasks in the main application is preserved in the model. For example, a process should become ready only after some interesting event is triggered.

All these constraints considered, we aim to provide a modeling schema for translating PicOS applications to corresponding NuSMV models. For instance, the NuSMV model of the application presented in Figure 3.1 and Listing 3.1, is as illustrated in Listing 3.3. According to the state-machine notion of NuSMV, the model is the combination of variables that may change their value at each step and change the state of the system respectively; therefore, the behavior of the model is resulted by the decisions made for each variable at each step.

As presented in Listing 3.3, the extracted model contains of multiple modules (i.e. modeling elements) each of which are responsible for modeling a cohesive behavioral aspect of the application, either in the defined entry level or in fundamental kernel or network layers. Details of what each modeling element is responsible for, and how its functionality can be extracted from the PicOS application are explained in following sections. However, before jumping into detailed modeling procedure, we present a high level behavioral analysis of PicOS systems and the logical layered architecture to be modeled.

```
MODULE main
  VAR
    interrupt : word[5];
    NODE1 : node1(interrupt);
  ASSIGN
    init(interrupt) := 0b5_0;


MODULE node1(interrupt)
```

```
  VAR
    id : 0..100;
    event: word[10];
    pcb : pcb(id, event);
    s : shared(pcb , event, interrupt);
    sch : scheduler(id, pcb);


MODULE scheduler(currentID , pcb)
  ASSIGN
    init(currentID) := 0;
    next(currentID) :=
      case
        next(pcb.lock) : currentID;
        pcb.lock : 0;
        TRUE : (currentID+1) mod 1;
      esac;


MODULE pcb (id, event)
  DEFINE
    lock := RED.lock | GREEN.lock;
  VAR
    sample : sampleProcess(0, id, event);


MODULE shared(pcb , event, interrupt)
  VAR
    internal : word[5];
  ASSIGN
    init(internal) := 0b5_0;
    next(internal) :=
      case
        next(pcb.sample.state) = S1 : 0b5_01; //trigger event E1
        TRUE : 0b5_0;
      esac;
    event := interrupt::internal;

MODULE simpleProcess(pid, id, event)
  DEFINE
    lock := state=s1 | state=s2 | state=s3 | state=s4;

  VAR
    pevent : word[10];
    pstate : array 1..10 of {s1, s2, s3, s4};
    state : {s1, s2, s3, s4, wait, ready};
    nextstate: {s1,s2,s3, s4};
    timer : 0..1000;
    tstate : {s1,s2,s3,s4};

  ASSIGN
    init(state) := ready;
    init(nextstate) := s1;
    init(pevent) := 0b10_0;
    init(timer) :=0 ;
    next(pevent) :=
      case
        state=wait & ((event & pevent) != 0b10_0| timer=1) = 0
            b10_0;
        state = s1 : 0b10_111;
        state = s2 : 0b10_11;
        TRUE : pevent;
      esac;
    next(pstate[1]) :=
      case
        state = s1 : s2;
        state = s2 : s1;
        TRUE : pstate[1];
      esac;
    next(pstate[2]) :=
      case
        state = s1 : s1;
        state = s2 : s4;
        TRUE : pstate[2];
      esac;
    next(pstate[3]) :=
      case
        state = s1 : s3;
        TRUE : pstate[3];
      esac;
    next(timer) :=
      case
        state = wait & ( (event & pevent) != 0b10_0 | timer=1) :
            0;
        state = s1 : 100;
        state = s2 : 50;
        timer != 0 : timer - 1;
        TRUE : timer;
      esac;
    next(tstate) :=
      case
        state = s1 : s3;
        state = s2 : s4;
        TRUE : tstate;
      esac;
    next(nextstate) :=
      case
        state=wait & (event & pevent & 0b10_1)!=0b10_0 : pstate
            [1];
        state=wait & (event & pevent & 0b10_10)!= 0b10_0: pstate
            [2];
        state=wait&(event & pevent & 0b10_100)!= 0b10_0: pstate
            [3];
        state = wait & timer=1 : tstate;
        state = s3 : s2;
        TRUE : nextstate;
      esac;
    next(state) :=
      case
        state = wait & ( (event & pevent) != 0b10_0 | timer=1) :
            ready;
        state=ready & pid=id : nextstate;
        state=s1 : wait;
        state=s2 : wait;
        state=s3 : ready;
        state=s4 : s3;
        TRUE : state;
      esac;
```

Listing 3.3: NuSMV Model of Comprehensive PicOS Example

## 3.2 Analysis

In order to provide a modeling schema to abstract a set of applications, we should know about (1) the features of the source applications that are interesting to be abstracted and model checked afterwards, (2) the constraints of the modeling environment that prevent us from modeling those properties.

Our ultimate goal is to provide a comprehensive abstraction of PicOS applications. The resulting model should simulate the behavior of the original application executed on PicOS node. In other words, the observable results or status changes of the application must be reflected in the model execution. Therefore, the model should include all the properties that are essential to the process of PicOS application execution.

The execution of a PicOS application on a destination node also depends on the underlying behavior of the operation system. The application source file only contains the highest-level logic of the procedure, such as the processes of a node, the states they can have, etc. However, there are many rules defined in PicOS that control the workflow of the application. For example the kernel is responsible for scheduling processes to be executed, the inter-process communication through events, raising and handling events and etc.[2]

The behavior of a PicOS application is resulted from the visible requirements of the application as defined in the PicOS program, or comes from the kernel of the PicOS operating system. For instance, when a node is programmed to turn on/off a LED on recognizing an event, this blinking is directly resulted from the programmed application and can be visibly recognized. However, there are also PicOS kernel behaviors affecting on the way events can be recognized by processes, or the priority of the events in being handled.

Figure 3.2 illustrates the state diagram of a PicOS single-node application including two processes. This application is a two-LED blinker in which the LEDs blink alternatively. Each of the processes controls one LED and communicates to the other process for synchronization. Receiving

[2]Even beneath the operating system layer, the the hardware logic may affect the execution of the program, however, we do not consider those since the verification of hardware is beyond the domain of this thesis.

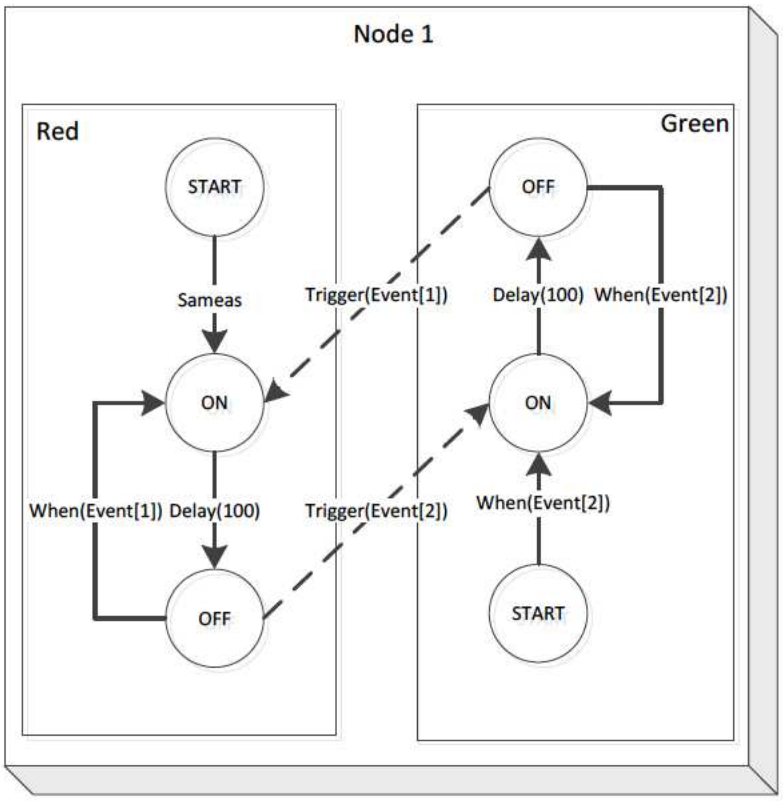

Figure 3.2: One Node 2 Process Blinker

an event from the other process, a process turns on its LED and after one second turns-off the LED and triggers an event awaited by the other process. The partial PicOS code of this application is presented in Listing 3.4. In this example, all the variable changes, state transitions, and reactions to triggered events are considered as visible requirements that should be considered while modeling such application. However, the way the events are prioritized for each process, the underlying structure for controlling for which events each process is waiting at any time, and also scheduling the processes to run only one at a time are the kernel behaviors which are not visible from the PicOS defined FSMs.

```
#include "sysio.h"
#define LED_GREEN 1
#define LED_RED 2
#define RED_EVENT (&red)
#define GREEN_EVENT (&green)

byte red, green;

fsm RED() {
  entry RED_START:
    sameas(RED_ON);

  entry RED_ON:
    diag("red %d", 1);
    leds(LED_RED, 1);
    delay(1024, RED_OFF);
    release;

  entry RED_OFF:
    diag("red %d", 0);
    leds(LED_RED, 0);
    trigger(GREEN_EVENT);
    when(RED_EVENT, RED_ON);
    release;
}

fsm GREEN() {
  entry GREEN_START:
    when(GREEN_EVENT, GREEN_ON);
    release;

  entry GREEN_ON:
    diag("green %d", 1);
    leds(LED_GREEN, 1);
    delay(1024, GREEN_OFF);
    release;

  entry GREEN_OFF:
    diag("green %d", 0);
    leds(LED_GREEN, 0);
    trigger(RED_EVENT);
    when(GREEN_EVENT, GREEN_ON);
    release;
}

fsm root() {
  entry ROOT_INIT:
    runfsm RED;
    runfsm GREEN;
    release;
}
```

Listing 3.4: PicOS For One Node 2 Process Blinker

On the other hands, Figure 3.3 illustrates the same ping pong blinker between two nodes. This example is about a multi-node blinker application. This application is similar to the application used in the previous example except that the processes are placed in two separate nodes. Therefore, the processes should use inter-node communication instead of intra-node communication for synchronization. We do not provide the PicOS representation of this application here. The reason is that, the inter-node communications in PicOS applications need details about message passing between the nodes (e.g. the structure of the packets) which is too complex to be interesting for our discussion. However, ignoring the complexities in implementing their PicOS programs, we also aim to model the multi-node applications to be able to check their intended behavior. Same as the previous

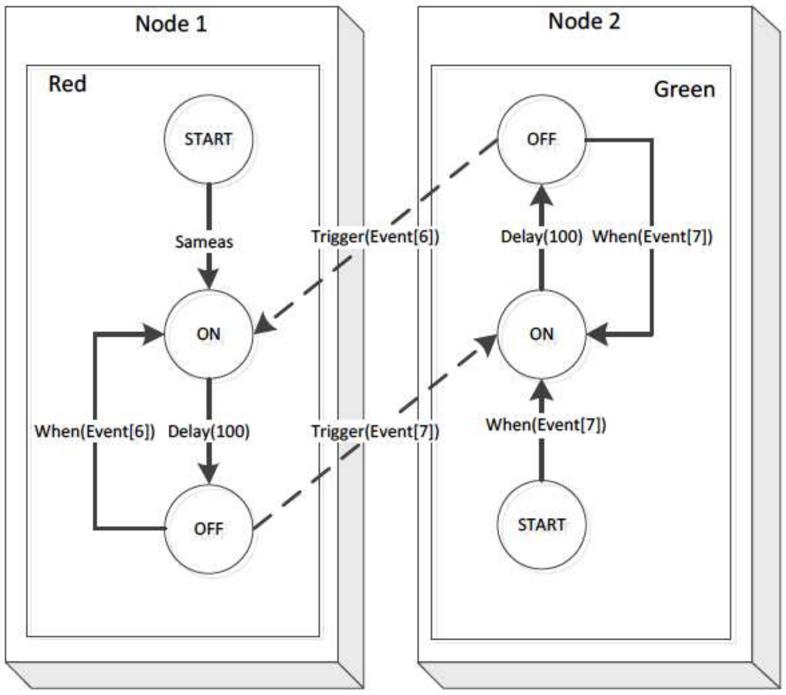

Figure 3.3: 2-Node 1-Process Blinker

example, the way inter-node communications are handled in such applications are not visible from application program (application logical layer) but are required. For instance if there is no protocol for passing messages between nodes, the processes waiting for an event which could only be triggered by another node can never proceed and provide their intended visible behavior.

The visible requirements are resulted directly from the programmed applications, whereas the kernel behavior may be differently simulated in SIDE-based PicOS simulators. Since our observations are based on behavior presented by such emulators, we need to base our model on only one of the possible ways of emulations [3]. Therefore, except for detecting the visible and kernel behaviors to be modeled, we should clarify how the underlying operating system (here the emulator) should coordinate the application.

All these considered, we need to make some assumptions on the PicOS properties to be modeled before designing the modeling schema. It is worth noting that, simplifying assumptions are made to omit the non-deterministic behaviors of the system and provide an easy to justify basis for PicOS applications. Based on these assumptions, we will know what the intended model should do and we can also validate our model accordingly; therefore, we can build a trustable modeling schema that can be used for verifying complex examples.

[3]Our goal is to see if modeling PicOS applications are feasible and we can verify them. Therefore, covering all possible coordination alternatives of an application is beyond our goals.

### 3.2.1 Modeling Assumptions

Our ultimate goal was to provide a modeling schema that can transform as wide variety of PicOS applications as possible. However, for many reasons it is not possible to model all properties of a PicOS application precisely. For example, NuSMV simply does not support a notion of time (even in discrete steps), or complex operations on data types. Modeling this is possible but results in a model that is too complex to be model checked. On the other hand, there is no single behavior of PicOS. For instance, the process scheduler that is responsible for selecting the process to run may work using a Round Robin strategy or a priority queue based on the order of defined processes in the program, or the exact time a triggered event can be realized by other processes. Therefore, we needed to select a subset of the PicOS context, in order to make the modeling possible while keeping the original behavior of the application.

We made some assumptions regarding modeling PicOS applications. These assumptions either limit the PicOS applications to those ones that can be modeled or specify the notable differences between the PicOS applications and their corresponding models[4]. As a result, we aim to provide a modeling schema for the PicOS applications that can satisfy the assumptions either originally or after some modifications.

Following is the list of the assumption we made for PicOS applications that can be modeled using our schema:

- The PicOS application should not have any syntactic or semantic errors. It means that the application should be compilable without any error. This way we can use a very simple translator from PicOS to NuSMV models.
- As our focus is on modeling the execution sequence of states, we are interested in the elements which influence the execution of states, not the elements which are related to manipulation of data. In other words, we model execution-control elements such as events, interrupts, when, delay, proceed, trigger, and sameas, while we ignore data-manipulation elements. As a result, execution-control elements should not generally have dependency to data-manipulation elements. For example, we cannot have a delay statement whose time value is decided by a PicOS function. However, some simple data-manipulation elements such as arithmetic operations and conditional statements can be used, but we do not guarantee to support them. Therefore, we should only use them with considering how to model them.
- The root module of every node should have exactly one state in which other processes are instantiated. The root module should not have any statement other than process instantiation ones.
- The root module can instantiate up to 100 processes.
- We assume that the later a process is instantiated in a PicOS application, the more priority it

[4]It is worth noting that, some of the assumptions are based on the observed behavior of PicOS SMURPH-based emulator.

will have.

- No other module except the root module should have process instantiation statements. As a result, all the processes are instantiated by the root module when a node starts and no other process is instantiated after that.
- Each state should explicitly issue the release statement at the end of the state unless it issues sameas or proceed statements.
- There should not be more than one delay statement in a state.
- There should not be more than one when statement with same event in a state. (Just to add that we can have multiple when statements with different events or even the combination of a delay statements with multiples when statements)
- When a sameas or proceed statement is used in a state, no other sameas, proceed, delay, and when statement should be issued.
- Processes in a node can communicate with each other just by issuing when and trigger statements. In fact, the only dependencies between processes in a node are through events. In addition, global variables can be defined and manipulated by processes.
- There can be up to 5 different event types in a process.
- Processes in different nodes can communicate with each other just by waiting for packets and sending packets. In fact, the content of the packets is not important and the packets are just used as interrupts.
- There can be up to 5 different interrupt types in a process which can be used for either communication between processes in different nodes or receiving various signals from the hardware.
- We cannot model time in the same way as PicOS. Therefore, we use the execution steps of NuSMV as our time. Therefore, the modeled time is discrete as opposed to continuous time in PicOS.
- We assume that every state is atomic and runs in one step.
- Interrupts and internal events can only be triggered once in one step and are received by the running state in the beginning of its execution.
- A delay statement can have a time value up to 1000, corresponding to 1000 steps event transitions.

Many of these assumptions are parameters, and can be adjusted as needed. Others are deeply threaded through the model.

## 3.3 Architecture Model

Having all modeling requirements and limitations explored in the previous phase, we need to think of the architecture and design of the model systematically. The broad range of functional and nonfunctional requirements that are intended to be satisfied by the modeling schema is the reason the design phase is the most crucial step of the modeling procedure. For instance, the modeling schema covers a subset of the PicOS properties limited by modeling assumptions; therefore, we should consider extensibility of the schema for future extensions if needed. In addition, the obtained models should be understandable for someone who wants to model check the program specification; therefore, it is needed to be compatible with the PicOS logical layered representation, e.g., with equivalent parts for network, processes, and etc.

We need to consider different abstraction layers in modeling PicOS applications. There are two fundamental reasons for that:

1. The broad domain of PicOS logic requires a structured model to cover all the needed properties. On the one hand, in verifying a single-node PicOS application, all rules regarding CPU allocation for each instance of a FSM, interaction between the FSMs, and event handling, need low-level modeling considerations based on PicOS kernel. On the other hand, the model needs to be extended to verify multi-node PicOS applications in which we have inter-node communications for sending and receiving network packets.
2. The logic needs to be decomposed and encapsulated in cohesive components so that the visibility of them can be limited. Not every property of the application units is public to all other components. For example, the behavior of a process is not visible by other processes in the node and the processes can only communicate through the event interfaces; however, each node should know about all of its processes. Moreover, the interior design of node is not accessible by other nodes in the network and their communication channels are interrupts.

The underlying code for PicOS applications should be decomposed in cohesive components, and the required communication channels between components should have specific interfaces. Each component can access its own properties and those that have been passed to it through the interfaces. These components and interfaces, altogether, build the architecture of our modeling system.

As the first step in presenting the architectural model of our modeling schema, we decompose the functionalities of the applications, and demonstrate a layered logical architecture of the system. Figure 3.4 illustrates two major levels of the applications: *Node* layer and *Network* layer. The node layer itself could be decomposed into *kernel* layer and *application* layer of a single node application.

Node layer contains every piece that is needed for a single node application. Within node layer, the user defined application consisting of the PicOS commands, e.g. wait(event, state), delay(time, state), and the underlying coordinative kernel behavior should be modeled. A user-defined program, a.k.a application layer, is the highest layer of abstraction we have in our model. At this level, the only concern is to have a well written program (or model) that assumes all the underlying layers work properly, e.g., assuming that scheduler tells the processes when to "run" from a lower level, each process can receive the "run" message from scheduler and become activated if ready. Therefore, on this layer, our concern is to translate the PicOS commands to state transitions and facilitate the communication between the application and kernel layers.

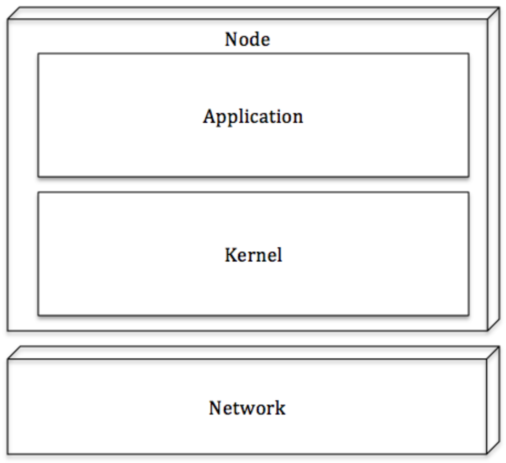

Figure 3.4: Conceptual PicOS Architecture

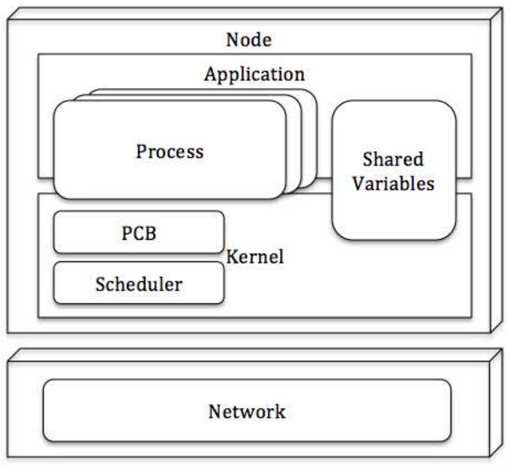

Figure 3.5: PicOS Modeling Components

Figure 3.6: PicOS Architectures

Kernel layer, on the other hand, contains the operating system rules and all the structure needed for managing processes of a single node, e.g., scheduling processes to execute, handling timer properties and triggering time event, providing a channel for communicating with upper layer (application layer) and other nodes (network layer).

Network layer is used in multi-node applications and is responsible for defining the network topology and the inter-node communications. Therefore, the network graph of the system should be build in this layer; the nodes should be instantiated as the graph vertices, and the communication channels between nodes should be organized as edges in the graph.

## 3.4 Component-based Design

Knowing about the logical decomposition of the application, presented in Figure 3.4, we can select the major model components of the system, so that (1) each of the components is responsible for a set of cohesive functionalities, and (2) their combination can cover the whole network and node level properties. Figure 3.5 demonstrates the components that are included in the model. Each component is responsible for providing a set of functionalities and covers some parts of the proposed logical architecture. The fundamental modeling elements of the proposed schema and their functionalities are discussed in the following subsections. We introduce all these modeling elements with a top down approach, starting from process that includes the major logic of the application layer, continuing with inner logic of node layer that resides in the node, scheduler and PCB elements, to the lowest level logics in network to manage inter-node links and message passing.

This component based design suffices to show the modeling approach and demonstrates its potential in covering behavioral aspects of PicOS kernel and applications. Therefore, we only present the functionality provided by each component to present the big picture to the reader. The detailed design and implementation guidelines for extracting NuSMV models from PicOS applications are presented in Appendix A.

**Process**

A process is the building block of any PicOS application. It is essentially a user defined FSM. Each application consists of a number of processes that define the behavior of the node. In our modeling schema, we opted for processes not only to include the high level behaviors that directly come from the PicOS commands, such as state transitions for proceed and sameas, but also to partially cover the low level behavior that comes from the internal logic of the PicOS kernel, such as updating its state to ready whenever an interesting event is triggered and flushing the list of events for which the process is waiting. In other words, our process units provide an abstraction of the FSM coded for the application and cover some basic behavior that is a PicOS kernel responsibility. Referring to the layered logical architecture illustrated in Figure 3.5 the process element covers the most of the application layer and part of the Kernel layer. Our modeled process provides the following functionalities:

1. Process is responsible for saving its state: ready, wait or one of the active states defined in the related FSM; and for changing its state according to the rules.
2. Whenever in the wait state, a process should have a list of events that it is waiting for, and change its state to ready if one of the events occurs.
3. Process is capable of recognizing what events have occurred.
4. Process is responsible for keeping a timer for tracking time and triggering time event according to the rules.
5. For each event that the process is waiting for, in the wait state, it should know at which state it should wake up if the event occurs.
6. Process should know if it is the one scheduled to have the CPU next, and it should be activated if it is its turn.

**PCB**

PCB (Process Control Block) is a structure for keeping all instances of processes defined for a single node. It models the PCBT in the PicOS kernel and contains a list of processes. In a PicOS application that can be modeled with our schema, PCB can also act as the root FSM of PicOS application in which just the instantiation of all processes occurs. The PCB data structure facilitates the usage of process data for other elements in the model:

- It provides a single point of access to the processes. If any element in the model, such as the scheduler or sharedVariables, needs to know about any of the processes or any data residing in a process, it can use the PCB as an interface to access the processes. Any element that has access to the PCB, has access to all the processes that are defined in the node.

- It contains a list of processes from which the scheduler can select the next one to run.
- It controls the execution of processes by informing the scheduler about which process is running, and by informing processes about whose turn it is to get the CPU. Each time one of the processes is running, the scheduler is informed so that it can start over and iterate from the beginning of the list of processes. Whenever the scheduler allocates the CPU to one of a process, the process id is be announced to all the processes, so that the others stop running and only that process runs.

**Scheduler**

The scheduler is the controller core of every node since it decides which process of the node can use the CPU at any time. According to our modeling assumptions (Section 3.2.1) the scheduler should start iterating from the beginning of the process list (PCBT) each time a process stops running. An example of a scheduling scenario with 3 processes is as follows:

**step 1:** scheduler tells p1 to work,
  if p1 is ready, it gets the CPU and runs, after its turn ends GoTo step 1.
**step 2:** scheduler tells p2 to work,
  if p2 is ready, it gets the CPU and runs, after its turn ends GoTo step 1.
**step 3:** scheduler tells p3 to work,
  if p3 is ready, it gets the CPU and runs, after its turn ends GoTo step 1.
  else GoTo step 1

In our modeling schema, this is the strategy that the scheduler is responsible to perform. However, the strategy used by scheduler may vary in different versions of PicOS kernels. Therefore, for model checking the behavior of the application using different schedulers, the implementation of the scheduler module in the model should match the one that is running on the destination nodes. Accordingly, by modifying the strategy of the scheduler, the schema can be used for those different versions of PicOS too.

Note that one of fundamental responsibilities of the PicOS scheduler is to make sure that no two processes are running at the same time. This responsibility is partially modeled in each process by comparing the identifier of the current running process (currentId) with the process id (See Section A.1 in Appendix A). Accordingly, part of scheduler responsibilities is scattered in other parts of our model.

**SharedVariables**

As it was explained in Section 3.1.2, for each defined variable in the model we can assign value to it only in one module. In other words, we can have the reading access to a variable in more than one module, but the writing access on that variable is restricted to one of the modules. On the other hand, in modeling PicOS applications, there may be some variables shared between processes, e.g. event/interrupt variable, and we need to change the value of the shared variables based on the processes status, e.g. each process wants to trigger a subset of events if it is in some specific active state. For example, we want to trigger the first event if the state of p0 (an instance of a process) is s1 or the state of p1 (another instance of a process) is s2, and we want to trigger the second event if the state of p1 is s2. The value that should be assigned to the variable event depends on different internal variables of processes; but we cannot write the assignment conditions in each of the processes individually, since there should be only one module assigning values for each variable. Therefore, this challenge of "where should the values be assigned to the variable" exists for every variable which is shared between multiple modules or instances of modules; here the shared variable is event which is shared among instances of processes.

To solve this problem, we defined the sharedVariable module that is responsible for organizing modifications of the variables that are shared between processes, e.g. event.

**Node**

Node is the complete executable application of a single node. It should contain all logic in an actual PicOS node, both the application layer and kernel level. The node is the structure that combined all of the kernel and process modules into one model. Each node should be capable of communicating with other nodes in a multi-node application. Following is the list of responsibilities required to be facilitated by node element:

- Node element is responsible for instantiating unique instances of scheduler, PCB and share-Variable elements.
- To facilitate inter-node communication, node should have an interface, say an interrupt variable, to be able to use as a broadcasting channel in network level communications.

**Network**

Network contains a set of nodes that can communicate with each other using interrupts. In other words, interrupts are the communication channels between nodes. Therefore, if we have a set of nodes using an interrupt for communication, every node that has access to that interrupt can be assumed to be connected to the network of those nodes. That is, we can consider interrupts to be a shared variable between a set of connected nodes. So, the interrupt can be changed based on the internal logic of the nodes sharing it, every change to the interrupt can be seen by nodes, and every node in the network which is sharing that interrupt can react to the changes in interrupts. This also means that interrupts can be used for broadcasting messages to nodes which are in range, i.e. nodes which share that interrupt.

Accordingly, we need our network to instantiate the nodes and connect them using interrupts. In addition, since interrupts are assumed as shared variables between nodes, we need to organize interrupt modifications in the network. Following is the list of functionalities for which the network element is responsible:

- As the structure knowing about all inner nodes, network should create the graph of the nodes in the multi-node applications.
- To facilitate inter-node communication, network should provide an interface to nodes, say an interrupt variable. Organizing the interrupts, as the shared variable among all nodes in the application is the network's responsibility.

## 3.5 Testing The Modeling Schema

After modeling any system (e.g. creating the components based on the proposed architecture and applying the corresponding modeling guidelines provided in Appendix A), the next inevitable step is to confirm the correctness of the model, and check if the model is a right abstraction of the original system. The reason is that we cannot prove anything about the main object (here the PicOS application) in model checking procedure unless we know that the behavior of the model is the same as the behavior of the main system. Therefore, since the focus of our modeling is on abstracting the scheduling of processes and their state transitions based on PicOS logic, we should check if the execution sequence of states in our model is the same as the execution sequence of states in the PicOS application.

In order to check the correctness of our model, and see if our modeling guidelines can result to a model compatible with the PicOS application, we should start with a PicOS application that we exactly know how it behaves regarding the sequence of states being executed and the schedule of the processes of any node in the application. Then, we should model this application using our proposed modeling guidelines. Therefore, we will have a model that is desired to work the same as the main application. To check if the model meets our expectations, we should write the expected behaviors as specifications. If our model is correct, it should show that the specifications are true. Otherwise, if the model checker provides a false answer for any of the specifications, it means that the procedure of modeling of the application should be revised since one of the followings might have happened:

- We have had wrong expectation from the PicOS application.
- The expected behavior has not been expressed correctly as specification.
- We had not followed the guidelines properly and that PicOS application has not been modeled as desired.
- The provided guidelines are faulty.

Among the above mentioned items, we are mainly interested in the fourth one since our goal is to verify the proposed modeling guidelines. Therefore, if we find any case of faultiness about the guidelines, we should go back to our modeling procedure to find the source of faultiness and revise our guidelines.

Note that applying this approach, we may not provide a thorough verification of our model. The reason is that checking all possible ways of state transitions and process scheduling for the model and the application is tedious and beyond the available time for this project.

To provide an example of how this approach can be employed for checking the correctness, here we explore a simple single-node application in PicOS, model it according the main application and the provided guidelines, and then check the model against some specifications based on the behaviors we know can be revealed by the PicOS application to make sure if the model and the PicOS application are compatible to each other.

Figure 3.7 and Listing 3.5 show the state diagram of a single-node application and its PicOS representation. This application is a simple one that contains a process with three running states, namely START, S1 and S2. The process starts running in state START, goes to S1, and then cycles in states S2 and S1 in a forever loop. The point in this cycling is that, for performing the transitions which are labeled with proceed, CPU is released while the process remains ready to start running in the destination state; whereas for performing the transitions which are labeled with sameas, the process directly jumps to the destination state without releasing the CPU in between.

```
#include "sysio.h"

fsm test1() {
  entry START:
    diag("1");
    proceed(S1);

  entry S1:
    diag("2");
    sameas(S2);

  entry S2:
    diag("3");
    proceed(S1);
}

fsm root() {
  entry INIT:
    runfsm test1;
    release;
}
```

Listing 3.5: PicOS Code For 1-Node 1-Process Validation Test

Based on the intended PicOS application and by following the provided guidelines for modeling the application, we create the model which is the abstraction of the application in NuSMV. Listing

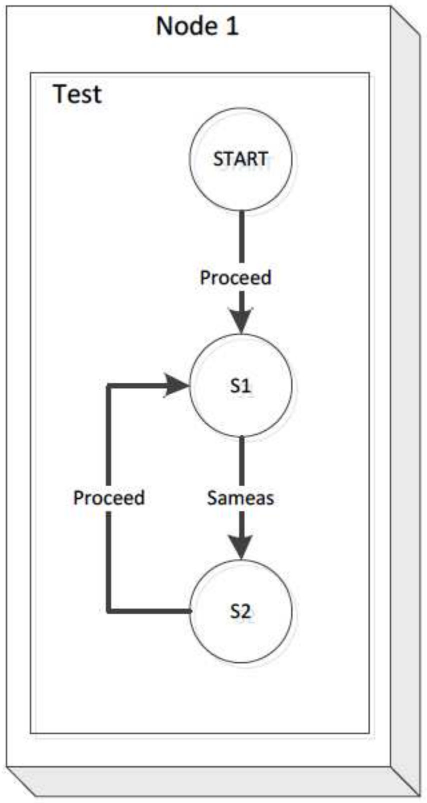

Figure 3.7: 1-Node 1-Process Validation Test

3.6 shows the created model of the PicOS application. Having the model, the next step is to check the compatibility of the application and the model with respect to some known features of the PicOS application.

```
MODULE main
  VAR
    interrupt : word[5];
    NODE1 : node1(interrupt);
  ASSIGN
    interrupt := 0b5_0;


MODULE node1(interrupt)
  VAR
    id : 0..100;
    event: word[10];
    pcb : pcb(id, event);
    s : shared(pcb , event, interrupt);
    sch: scheduler(id, pcb);

MODULE scheduler(currentID , pcb)
  ASSIGN
    init(currentID) := 0;
    next(currentID) :=
      case
        next(pcb.lock) : currentID;
        pcb.lock : 0;
        TRUE : (currentID+1) mod 1;
      esac;


MODULE pcb (id, event)
  DEFINE
    lock := Test.lock;
  VAR
    Test : test(0, id, event);


MODULE shared(pcb , event, interrupt)
  VAR
    internal : word[5];
  ASSIGN
    internal := 0b5_0;
    event := interrupt::internal;

MODULE test(pid, id, event)
  DEFINE
    lock := state=S1 | state=S2 | state=START;
  VAR
    pevent : word[10];
    pstate : array 1..10 of {START, S1, S2};
    state : {START, S1, S2, wait, ready};
    nextstate: {START, S1, S2};
    timer : 0..1000;
    tstate : {START, S1, S2};
  ASSIGN
    init(state) := ready;
    init(nextstate) := START;
    pevent := 0b10_0;
    timer :=0 ;
    next(nextstate) :=
      case
        state = START : S1;
        state = S2 : S1;
        TRUE : nextstate;
      esac;
    next(state) :=
      case
        state = wait & ( (event & pevent) != 0b10_0 | timer=1) :
            ready;
        state=ready & pid=id : nextstate;
        state=START : ready;
        state=S1 : S2;
        state=S2 : ready;
        TRUE : state;
      esac;
```

Listing 3.6: NuSMV Model For 1-Node 1-Process Validation Test

Based on what is illustrated in Figure 3.7 and Listing 3.5 we chose five behavioral features about the execution sequence of the states in the application. We first made sure about the correctness of the expectations by testing the PicOS application. Then, we wrote the specifications for these behavioral expectations for which we want to check our model. Followings are the list of the expectations and their specifications.

- The process never goes to sleep (wait state).
  $SPEC\ !EF(NODE1.pcb.Test.state = wait);$

- If the process is in state S1, it will always go to state S2 next, even without releasing the CPU.
  $SPEC\ AG(NODE1.pcb.Test.state = S1 \rightarrow AX(NODE1.pcb.Test.state = S2));$

- If the process is in state START or S2, it releases the CPU and becomes ready immediately.
  $SPEC\ AG((NODE1.pcb.Test.state = START \mid NODE1.pcb.Test.state = S2) \rightarrow AX(NODE1.pcb.Test.state = ready));$

- The process will eventually reach a situation after which its state will never change to START again.
  $LTLSPEC\ F(G(NODE1.pcb.Test.state! = START));$

- The state of the process will eventually become state S1, and after that it will cycle in states S2 and S1.
  $LTLSPEC\ F(NODE1.pcb.Test.state = S1\ \&\ F(NODE1.pcb.Test.state = S2\ \&\ F(NODE1.pcb.Test.state = S1)))$

If our model is compatible with the main PicOS application, the model checker will show that all the above mentioned specifications are true. If not, we should go through the features, specification, modeling steps and guidelines to see what causes the incompatibility.

This example is just a very simple one which could show the correctness of our modeling approach for this single process application. However, in order to enhance our confident about the correctness of our modeling guidelines in modeling any kind of PicOS application, we need to do the same steps for more complex applications with more processes and examine various specifications on the models.

We designed a set of test applications and specifications and used them to verify our guidelines based on our modeling assumptions. We checked the consistency of model and intended PicOS kernel behavior by trying the following problems [5].

- At most one process should be running at a time.

[5]For each of these problems many test cases were designed and checked against specifications. The more test applications and specifications we examine, the more we become confident about the correctness of our modeling schema. However, we decided not to include the detailed test cases for each problem since it would take too many pages.

- The priority of the processes for being scheduled are according to their order in PCB; the later a process is instantiated in a PicOS application, the higher priority it will have.

- Processes in a node can communicate with each other just by issuing “when” and “trigger” statements.

- All processes waiting for an event should be notified when the event is triggered.

- The execution of each running state should take only one step.

- Each Process should stay locked (running) only for one step unless its running state contains the sameas statement (resulting to a jump to another running state and keeping the lock for one more step).

- When multiple events and/or interrupts occur at the same time, each process handles at most one of them. If the process is not waiting for those events it ignores them all, if it is waiting for one or more of them, it picks the one with the lowest corresponding bit [6].

## 3.6 Applying The Schema For PicOS Application Analysis

In the previous section, we defined different specifications for checking the correctness of our model and making sure that when we model a PicOS application, the model and the main application behave the same way. In this section we want to benefit from the correctness of our models, by using our model in verifying PicOS applications.

Employing our modeling approach for PicOS systems verification is somehow the opposite of what we did for checking the correctness of modeling guidelines. For verifying the modeling guidelines in the previous section, we had PicOS applications and we wanted to prove that the derived models are compatible to the main applications. But in this section, we assume that our modeling guidelines are correct and behave the same as the modeled applications, and we want to employ it to check if a PicOS application meets some desired features or not. For this purpose, we create the NuSMV model of the application and model check the model of the PicOS system. Since modeled behavior is the same as the original PicOS behavior, the PicOS application is correct if and only if the specifications (i.e. the expected behaviors) are evaluated to true for the model.

In order to show how we can employ our modeling solution for verifying PicOS applications, we explore a sample PicOS application in this section. Our sample is adopted from an example in PicOS documentation [3]. The sample is a single node PicOS application which has two processes named One and Two. The processes want to play a ping-pong game with signals in which One triggers an event awaited by Two ant Two replies by triggering an event awaited by One, and so on. The state diagram of the application is illustrated in Figure 3.8.

[6]In NuSMV model each event and interrupt is assigned a bit and their priority is decided based on to which bit of the events they have been assigned.

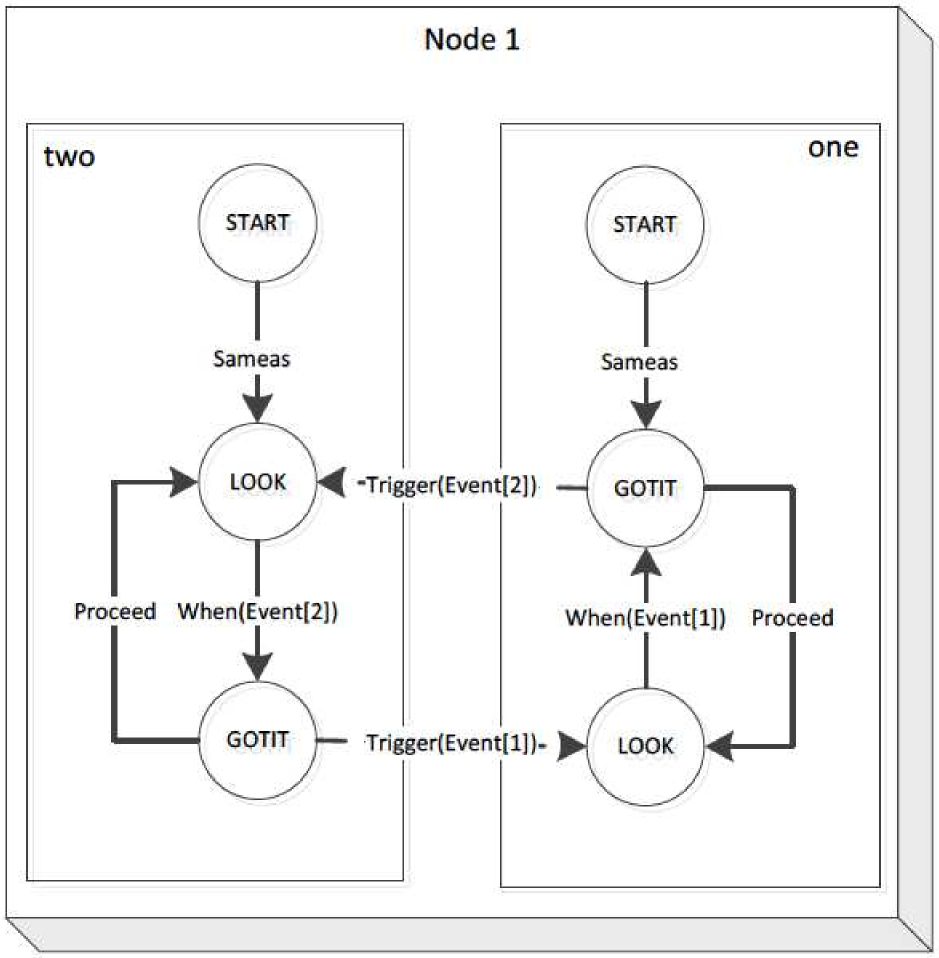

Figure 3.8: Example With Error

Listing 3.7 contains the implementation of the the application in PicOS language. In this application, each of the processes has three entries named START, LOOK and GOTIT. Receiving the expected event, a process wakes up in GOTIT entry, triggers the event awaited by the other process and then proceeds to LOOK entry. In the LOOK entry, the process waits for the expected event to go again to GOTIT entry. These steps are taken by both of the states to play the game.

```
#include ?sysio.h?
#define EVENT1 (&red)
#define EVENT2 (&green)
byte red, green;

fsm one() {
entry ONE_START:
  sameas(ONE_GOTIT);
  entry ONE_LOOK:
  when (EVENT1, ONE_GOTIT);
  release;
entry ONE_GOTIT:
  trigger (EVENT2);
  proceed(ONE_ LOOK);
}

fsm two() {
  entry TWO_START:
    sameas(TWO_ LOOK);
    entry TWO_ LOOK:
    when (EVENT2, TWO_GOTIT);
    release;

  entry TWO_GOTIT:
    trigger (EVENT1);
    proceed (TWO_ LOOK);
}

fsm root() {
  entry INIT:
  runfsm one;
  runfsm two;
  release;
}
```

Listing 3.7: PicOS For Example With Error

In order to examine this application to see if it has the desired properties, we should first model the application in NuSMV. The model of the application according to our guidelines is presented in Listing 3.8. Having the NuSMV model, now we can write specifications for the properties we want to check.

```
MODULE main
  VAR
    interrupt : word[5];
    NODE1 : node1(interrupt);

  ASSIGN
    interrupt := 0b5_0;


MODULE node1(interrupt)
  VAR
    id : 0..100;
    event: word[10];
    pcb : pcb(id, event);
    s : shared(pcb , event, interrupt);
    sch: scheduler(id, pcb);

MODULE scheduler(currentID , pcb)
  ASSIGN
    init(currentID) := 0;
    next(currentID) :=
      case
        next(pcb.lock) : currentID;
        pcb.lock : 0;
        TRUE : (currentID+1) mod 2;
      esac;


MODULE pcb (id, event)
  DEFINE
    lock := ONE.lock | TWO.lock;

  VAR
    TWO : two(0, id, event);
    ONE : one(1, id, event);


MODULE shared(pcb , event, interrupt)
  VAR
    internal : word[5];

  ASSIGN
    init(internal) := 0b5_0;
    next(internal) :=
      case
        next(pcb.ONE.state) = GOTIT : 0b5_10;
        next(pcb.TWO.state) = GOTIT : 0b5_1;
        TRUE : 0b5_0;
      esac;
    event := interrupt::internal;


MODULE one(pid, id, event)
  DEFINE
    lock := state=GOTIT | state=LOOK | state=START;

  VAR
    pevent : word[10];
    pstate : array 1..10 of {START, GOTIT, LOOK};
    state : {START, GOTIT, LOOK, wait, ready};
    nextstate: {START, GOTIT, LOOK};
    timer : 0..1000;
    tstate : {START, GOTIT, LOOK};

  ASSIGN
    init(state) := ready;
    init(nextstate) := START;
    init(pevent) := 0b10_0;
    timer :=0 ;
    next(pevent) :=
      case
        state = wait &((event & pevent) != 0b10_0 | timer=1) : 0
           b10_0;
        state = LOOK : 0b10_1;
        TRUE : pevent;
      esac;
    next(pstate[1]) :=
        case
        state = LOOK : GOTIT;
        TRUE : pstate[1];
    esac;
    next(nextstate) :=
      case
        state=wait & (event & pevent & 0b10_1)!= 0b10_0 : pstate
           [1];
        state = GOTIT : LOOK;
        TRUE : nextstate;
      esac;
    next(state) :=
      case
        state = wait & ( (event & pevent) != 0b10_0 | timer=1) :
           ready;
        state=ready & pid=id : nextstate;
        state=LOOK : wait;
        state=GOTIT : ready;
        state=START : GOTIT;
        TRUE : state;
      esac;


MODULE two(pid, id, event)
  DEFINE
    lock := state=GOTIT | state=LOOK | state=START;

  VAR
    pevent : word[10];
    pstate : array 1..10 of {START, GOTIT, LOOK};
    state : {START, GOTIT, LOOK, wait, ready};
    nextstate: {START, GOTIT, LOOK};
    timer : 0..1000;
    tstate : {START, GOTIT, LOOK};

  ASSIGN
    init(state) := ready;
    init(nextstate) := START;
    init(pevent) := 0b10_0;
    timer :=0 ;
    next(pevent) :=
      case
        state =wait&((event & pevent)!= 0b10_0 | timer=1) : 0b10_0
           ;
        state = LOOK : 0b10_10;
        TRUE : pevent;
      esac;
    next(pstate[2]) :=
      case
        state = LOOK : GOTIT;
        TRUE : pstate[2];
      esac;
    next(nextstate) :=
      case
```

```
        state = wait&(event&pevent&0b10_10)!= 0b10_0 : pstate[2];
        state = GOTIT : LOOK;
        TRUE : nextstate;
      esac;
    next(state) :=
      case
        state = wait &((event & pevent) != 0b10_0 | timer=1) :
           ready;
        state=ready & pid=id : nextstate;
        state=LOOK : wait;
        state=GOTIT : ready;
        state=START : LOOK;
        TRUE : state;
      esac;
```

Listing 3.8: NuSMV For Example With Error

One of the interesting properties for this application is to check whether a deadlock may occur or not. We can write a specification for this property as follows:

$SPEC\ !EF(pcb.ONE.state = wait\ \&\ pcb.TWO.state = wait)$

This result of executing this specification on the model will show whether there is a state in this application in which both processes are in wait states or not. If the specification is evaluated to true, our application may trap in a deadlock situation since when both processes are waiting for the other process, no further events will be triggered and no change in the states of the processes will happen. If it is evaluated to false, no deadlock will occur in the application and the processes will play the ping-pong game as long as the application is running.

Running this specification, the NuSMV returns false value which indicates that a deadlock situation may happen when executing our application. In addition, the NuSMV returns a counter example which shows how the specified state is reached from the initial state. Using the counter example, we can find out by which scenario the application goes to the deadlock situation. The counter example for our specification is presented in Figure Listing 3.9.

```
-> State: 1.1 <-
interrupt = 0ud5_0
NODE1.id = 0
NODE1.event = 0ud10_0
NODE1.pcb.TWO.pevent = 0ud10_0
NODE1.pcb.TWO.pstate[1] = START
NODE1.pcb.TWO.pstate[2] = START
NODE1.pcb.TWO.state = ready
NODE1.pcb.TWO.nextstate = START
NODE1.pcb.TWO.timer = 0
NODE1.pcb.TWO.tstate = START
NODE1.pcb.ONE.pevent = 0ud10_0
NODE1.pcb.ONE.pstate[1] = START
NODE1.pcb.ONE.pstate[2] = START
NODE1.pcb.ONE.state = ready
NODE1.pcb.ONE.nextstate = START
NODE1.pcb.ONE.timer = 0
```

```
NODE1.pcb.ONE.tstate = START
NODE1.s.internal = 0ud5_0
NODE1.pcb.lock = FALSE
NODE1.pcb.TWO.lock = FALSE
NODE1.pcb.ONE.lock = FALSE

-> State: 1.2 <-
NODE1.pcb.TWO.state = START
NODE1.pcb.lock = TRUE
NODE1.pcb.TWO.lock = TRUE

-> State: 1.3 <-
NODE1.pcb.TWO.state = LOOK

-> State: 1.4 <-
NODE1.pcb.TWO.pevent = 0ud10_2
NODE1.pcb.TWO.pstate[2] = GOTIT
NODE1.pcb.TWO.state = wait
NODE1.pcb.lock = FALSE
NODE1.pcb.TWO.lock = FALSE

-> State: 1.5 <-
NODE1.id = 1

-> State: 1.6 <-
NODE1.pcb.ONE.state = START
NODE1.pcb.lock = TRUE
NODE1.pcb.ONE.lock = TRUE

-> State: 1.7 <-
NODE1.event = 0ud10_2
NODE1.pcb.ONE.state = GOTIT
NODE1.s.internal = 0ud5_2

-> State: 1.8 <-
NODE1.id = 0
NODE1.event = 0ud10_0
NODE1.pcb.TWO.pevent = 0ud10_0
NODE1.pcb.TWO.state = ready
NODE1.pcb.TWO.nextstate = GOTIT
NODE1.pcb.ONE.state = ready
NODE1.pcb.ONE.nextstate = LOOK
NODE1.s.internal = 0ud5_0
NODE1.pcb.lock = FALSE
NODE1.pcb.ONE.lock = FALSE

-> State: 1.9 <-
NODE1.event = 0ud10_1
NODE1.pcb.TWO.state = GOTIT
NODE1.s.internal = 0ud5_1
NODE1.pcb.lock = TRUE
NODE1.pcb.TWO.lock = TRUE

-> State: 1.10 <-
```

```
NODE1.event = 0ud10_0
NODE1.pcb.TWO.state = ready
NODE1.pcb.TWO.nextstate = LOOK
NODE1.s.internal = 0ud5_0
NODE1.pcb.lock = FALSE
NODE1.pcb.TWO.lock = FALSE

-> State: 1.11 <-
NODE1.pcb.TWO.state = LOOK
NODE1.pcb.lock = TRUE
NODE1.pcb.TWO.lock = TRUE

-> State: 1.12 <-
NODE1.pcb.TWO.pevent = 0ud10_2
NODE1.pcb.TWO.state = wait
NODE1.pcb.lock = FALSE
NODE1.pcb.TWO.lock = FALSE

-> State: 1.13 <-
NODE1.id = 1

-> State: 1.14 <-
NODE1.pcb.ONE.state = LOOK
NODE1.pcb.lock = TRUE
NODE1.pcb.ONE.lock = TRUE

-> State: 1.15 <-
NODE1.id = 0
NODE1.pcb.ONE.pevent = 0ud10_1
NODE1.pcb.ONE.pstate[1] = GOTIT
NODE1.pcb.ONE.state = wait
NODE1.pcb.lock = FALSE
NODE1.pcb.ONE.lock = FALSE
```

Listing 3.9: Counter Example For Example With Error

This deadlock is actually due to the behavior of proceed statements. When a process issues a proceed statement, it gives the other processes the chance to pre-empt the current process. Therefore, based on the priorities of the running processes, the current process may not get the CPU immediately. In our case, when process One issues the proceed statement in GOTIT entry to go to LOOK entry, process Two jumps in and gets the CPU. As a result, both of the processes will end up in LOOK entry which results in deadlock.

A solution for this problem is to use sameas statements instead of proceed statements. By changing the proceed statements to sameas statements, we will have a new program which can then be modeled in NuSMV. Running the same specification on the new model, we get true answer which means no deadlock will happen in the new application.

In this example, we selected a simple application and examined only one specification on it. As you may notice, discovering the deadlock in this application did not actually require this amount of work since the application has only one execution path and we can find the deadlock just by execution the PicOS application. However, the goal of this example was to show how the model checking process can be done on a PicOS application. For more complex applications which may have many execution paths, it is hard or even impractical to run the application and trace different paths manually. Therefore, using the approach we provided in this project can facilitate the process of examining various properties on PicOS applications.

## 3.7 PicOS Modeling Failure In Real Networks

The example provided in the previous section showed that modeling schema can be used for modeling PicOS applications which follows our modeling assumptions, and can also verify behavioral specifications on the model. However, our further attempts in modeling real sensor network PicOS applications (e.g. clustering algorithms [6]) revealed that the modeling schema fails in representing an appropriate abstraction of the PicOS networks properly with the intended behavior.

The original PicOS application, deployed on target networks, cannot be modeled properly, and the model cannot be an appropriate representation of the application. That is mainly because of our modeling assumptions and the different protocols PicOS uses for inter-node communications and specifically handling message conflicts. For instance, in a network of PicOS nodes, the nodes broadcast their messages (same as the bit-wise interrupts we considered for nodes) through radio signals; accordingly, each node is capable of distinguishing actual messages from noise signals. Besides, the occurrences of message conflicts can be detected and handled in some way, such as considering the conflicted messages as noises or selecting only one of them to be received. These are functionalities handled by physical and very low level network layers which are not considered in our modeling schema.

If a network application is needed to be actually verified, its single nodes applications should contain the lower level networking logics, such as the complex message passing protocols, packets prioritizing, and conflict handling strategies and all other lower level networking algorithm. The reason is that our modeling schema is capable of modeling applications for single nodes and can provide some bits for inter-node communications. However, by including such networking details, the size of the application to be modeled and the NuSMV model itself would increase drastically, the PicOS application would not be neat and understandable anymore, and the NuSMV model's decision tree needs a huge memory to be checked against each specification.

We figured the infeasibility of PicOS modeling while trying to model check the clustering algorithm [6] for a small network, three nodes connected as a triangle. Every single node should have broad cast its state using either of the two main messages: 1) saying if it thinks it is a clusterHead of a system, or 2) saying if the node is joined to another cluster, and the identifier of the head of its cluster. Even though the algorithm implemented was not complicated and the target network was simple the model got too complicated and infeasible to model check. The complexity of the application was resulted from the message passing protocols (sending appropriate packets, and analyzing the received messages), transmitting long packets (7 bits was needed, 1 bit for distinguishing the type of message, 2 bits for the ID of the source node, 2 bits for the ID of the clusterHead to which the node is joined), and scheduling the nodes for broadcasting in a conflict free environment in asynchronous network (using coin flips when more than one message was intended to be transmitted over the channels, and also distinguishing messages with specific patterns starts and ends to make sure those who are listening are not receiving exact number of bits (7 bits) for each message transmitted).

Accordingly, even for a small network of nodes running simple algorithms, the PicOS model is too large to be modeled and infeasible to be model checked. It will result to complicated applications and huge models, which even deciding about the right specification reflecting their interesting behavior is too hard. Consequently, in order to apply our approach for model checking sensor network applications we need a more basic infrastructure for deploying the applications, e.g. nodes, which we can program based on what can be modeled in NuSMV, and build a framework for its verification. Therefore, the result would be an infrastructure verifiable at each step of its construction, and could be extensible if the target extension could be modeled in the reasoning environment. This approach is pursued in this thesis and the framework is provided in the next chapter.

# Chapter 4

# Framework

Our PicOS study reveals that we cannot model check real PicOS application with all communication protocols and operating system details. The variety of possible paths while executing processes (based on the order of triggered events and messages or the priority of them in being handled) on the one hand, and the broad range of lower level system behavior in inter-node communications (handling packet losses and conflict on communication channels) on the other hand, made even the simplest PicOS systems too large to model check.

The challenge then becomes distilling the essence of a PicOS-like system into a form that can be feasibly model checked, and still has a useful actual implementation with hardware. This led to a simpler framework based on the following assumptions:

- All processes are pure finite-state machines.
- Within a processor, processes execute independently, and communicate via events.
- Processes are responsible for their own scheduling by triggering and handling events, and there is no specific kernel level operating system logic for prioritizing.
- Nodes (i.e. processors) can execute asynchronously, and can run at different speeds.
- Nodes can communicate over the network composed of binary channels consisting of lines (continuous channels) between digital input and output.

Just like PicOS runs on real hardware, we want our simplified model to also be executable on networks of small processing nodes, not the least so that we can observe the actual behavior of applications.

To achieve this goal, we selected Arduino microcontrollers that are flexible enough to be programmed for network-based applications in a PicOS FSM programming model. Arduino is an open-source electronics prototyping platform that is able to sense the environment by receiving input bits and can affect its surroundings by sending output bits. The Arduino boards include microcontrollers that can be programmed using C++. They are roughly as powerful as the nodes used to construct a PicOS-based sensor network.

Although radios are available for ArduinOS, we decided to work with a much simpler network: physical wires between digital inputs and outputs. This simpler Arduino-based system eliminates the complicated packet sending protocols and replaces them with simple communication channels that look like 1-bit shared variables.

In PicOS, the state transitions are programmed in C and look like normal C code. Converting them into NuSMV models requires reasonably powerful code analysis. We thus decided to further restrict the PicOS FSM programming language so that extracting models was easier. We created a domain-specific language (DSL) that enables one to write event-driven finite-state processes that communicate via events. Our major goal was to have a DSL that is translatable to both Arduino programs and their corresponding model for formal verification. This has the advantage of ensuring that both the executable code and the NuSMV model are consistent. We call this combination of DSL, model-checking, and executable code by the name ArduinOS framework.

We now present more details about the ArduinOS framework, the DSL and the translation procedures for extracting Arduino and NuSMV model of the original program.

## 4.1 ArduniOS Architectural Model

In the ArduinOS platform, PicOS properties have been simplified throughout all existing architectural layers. In this section, we explore the architecture of ArduinOS to see how the functionalities of each PicOS layer have been distilled in the new framework.

Recalling from Section 3.3, we defined the PicOS architecture including two main layers of node and network, where the node layer itself could be assumed as two separate layers of Application and kernel layers. While simplifying PicOS to create ArduinOS, we kept the same functionality modularization and hierarchy, while decreasing the level of details.

The application layer of the architecture contains the application programmed in a finite state machine style, almost identical to a PicOS application.

The kernel is heavily simplified. Since there is no actual operating system on Arduino devices, the kernel layer logic must be implemented as some application layer modules. That is, in ArduinOS there is no specific boundary between the application layer and the kernel layer. We have specific standard modules that take care of event handling and process triggering, with the following characteristics:

- Processes have slots in a process control block in the order in which they are instantiated.
- The scheduler iterates through the list of processes and schedules them for execution one at a time.
- When a process is waiting for a set of events, the first event that is triggered will set the corresponding next state for the process.

- If some events occur simultaneously, their handling priority is based on the order in the per-process waiting list.

The network layer contains all detail about internode communications. In the PicOS architecture, this layer is responsible for handling all message conflicts, packet passing, radio packets encoding and etc. In the ArduinOS platform we simplify this layer into a shared-variable style:

- Nodes communicate over the network composed of binary channels consisting of 1-bit lines (continuous channels) between digital input and output.
- Multi-bit messages are transmitted over parallel 1-bit channels.
- There are no interrupts that notify a processor about a change in the input channels, changes must be detected by polling.

## 4.2 ArduinOS Domain Specific Language

Our goal is to define a simple PicOS style programming language for writing applications for Arduino devices. The goal is to generate executable Arduino program along with their equivalent NuSMV models. We want our DSL to have the following properties:

- It should have an event-based notion.
- The applications should be stateful.
- It should provide the possibility of waiting for events and triggering them.
- Like PicOS, the application elements should include processes, states, and events.
- It should only support those basic data types (integer, bits, and booleans) that have corresponding models in NuSMV.
- Recursion and any nested method invocations that require a stack to execute are forbidden.
- It should be extensible for multi-node applications.
- It should have basic procedural control notions like conditional statements and while loops.
- It should be able to generate executable code and checkable models that are mutually consistent.

In general, it is quite difficult to translate arbitrary code into a NuSMV model, especially that is feasible to check. We cannot model dynamically sized variables such as a string, stack or list, they must have a fixed size. Only limited nesting of method calls is possible. All data structures have to be expressed at the bit level, with every additional bit potentially doubling the state space of the model. However, it is possible to convert a highly restricted and stylized subset of C into a model provided that we only work with data and controls that can be easily handled by a NuSMV model. Based on this approach, the ArduinOS DSL is proposed to include a PicOS-style subset of C of which applications are possible to be modeled thoroughly in NuSMV.

Recalling the example provided in Figure 3.2, Listing 4.1 presents the intended ArduinOS representation of the same behavior. The ArduinOS code is neat and easy to understand. Comparing the PicOS representation of the example (Listing 3.4) and its corresponding DSL program, it is easily obvious that ArduinOS DSL is based on the same programming style of PicOS and reuses some statements such as wait (a.k.a. when), delay and trigger. Besides that, it is simpler to understand since the events and outputs are defined explicitly and the outputs are single bits that their value can be easily observed by connecting LEDs to the output ports. We renamed some keywords to make them easier to understand, e.g. used *goto* instead of *sameas*, and changed *when* to *wait* but kept their semantic meanings; and also omitted some keywords and functionalities to make the language simpler, e.g. proceed keyword is not included in ArduinOS. Therefore, here we only focus on minimal required set of functionalities to model while guaranteeing that anything programmed in ArduinOS can be modeled in NuSMV using our proposed framework.

```
Node node1{

  output o_red;
  output o_green;

  event RED_EVENT;
  event GREEN_EVENT;

  process P_INIT{
    state S0{
      o_red =0;
      o_green =0;
    }
  }

  process RED {
    state RED_START{
      goto(RED_ON);
    }
    state RED_ON{
      o_red = 1;
      delay(1024, RED_OFF);
    }
    state RED_OFF{
      o_red = 0;
      trigger(GREEN_EVENT);
      wait(RED_EVENT, RED_ON);
    }
  }

  process GREEN {
    state GREEN_START{
      wait(GREEN_EVENT, GREEN_ON);
    }
    state GREEN_ON{
      o_green =1;
      delay(1024, GREEN_OFF);
    }
    state GREEN_OFF{
      o_green =0;
      trigger(RED_EVENT);
      wait(GREEN_EVENT, GREEN_ON);
    }
  }

  process root() {
    state ROOT_INIT{
      process P_INIT;
      process RED;
      process GREEN;
    }
  }
}
```

Listing 4.1: ArduinOS For 1-Node 2-Process Blinker

Listing 4.2 and 4.3 present another example revealing other dimensions of ArduinOS DSL Simplicity. This application provides similar behavior as Figure 3.3 but instead of being interrupt driven, it uses the polling strategy. The reason is that, each message communicated in ArduinOS are through bit-channels, and in order to be received, the destination node should be waiting for it, meaning that the node should check its input channel continuously to see if anything interesting has come. This approach makes modeling easier in the sense that nodes know what they are waiting for. Besides, because of using bits as messages, we will not have to consider the complexities about packet structures [1].

```
Node node1{

  output o_led;

  input i_chan_0;
  input o_chan_0;

  process P_INIT{
    state S0{
      o_led =0;
      o_chan_0=0;
    }
  }

  process RED {
```

[1]PicOS multi-node applications were hard to implement since the structure of the packets should have been set separately, and also make the corresponding NuSMV model drastically complicated.

```
    state START{
      goto(ON);
    }
    state ON{
      o_led = 1;
      delay(1024, OFF);
    }
    state OFF{
      o_led = 0;
      o_chan_0 =1;
      delay(1024, WAIT_FOR_INPUT);
    }
    state WAIT_FOR_INPUT{
      if (i_chan_0 ==0){
        delay(1024, WAIT_FOR_INPUT);
      }
      else{
        o_chan_0 =0;
        delay(1024, ON);
      }
    }
  }

  process root() {
    state ROOT_INIT{
      process P_INIT;
      process RED;
    }
  }
}
```

Listing 4.2: ArduinOS Code For 2-Node 2-Process Blinker (Node 1)

```
Node node2{

  output o_led;

  input i_chan_0;
  input o_chan_0;

  process P_INIT{
    state S0{
      o_led =0;
      o_chan_0=0;
    }
  }

  process GREEN {
    state START{
      goto(WAIT_FOR_INPUT);
    }
    state ON{
      o_led = 1;
      delay(1024, OFF);
    }
    state OFF{
      o_led = 0;
      o_chan_0 =1;
      delay(1024, WAIT_FOR_INPUT);
    }
    state WAIT_FOR_INPUT{
      if (i_chan_0 ==0){
        delay(1024, WAIT_FOR_INPUT);
      }
      else{
        o_chan_0 =0;
        delay(1024, ON);
      }
    }
  }

  process root() {
    state ROOT_INIT{
      process P_INIT;
      process GREEN;
    }
  }
}
```

Listing 4.3: ArduiOS Code for 2-Node 1-Process Blinker (Node 2)

### 4.2.1 Stateful Node Specification Language

**Processes** build the core of an AdruinOS application. A processor can have a number of processes each of which can have its own variables, and states. The states the programmer defines in the body of a process are called the *active states* of that process. In addition to the active states, the process has two additional implied higher-level states used for scheduling: *ready* and *wait*.

In the ready state, the process is ready to run and is waiting to be activated and change its state to one of the active states and performing the transition action defined in that state. If the process is in its wait state, it is waiting for one or more events to happen so that it can become ready. Accordingly, each process acts like a state machine that starts at the ready state, then become activated if it gets the CPU, and may go to wait or ready state based on the specified logic in the related active state. When the application starts, all of the processes are in ready state, and can change their state to the first defined state in their definition, which is the *start state*.

**States** specify the behavior of a process. Each state is a small code fragment that specifies the nature of the transition that the process will make. There are global per-node variables as well as per-process variables, which are available to all the states, or just the states in the process respectively. Each state also has its own local variables which are initialized every time the state is entered — thus they are not persistent over state transitions, and thus make the model smaller.

The code fragment is composed of simple assignment statements, if-else, and while. The result of a state transition is to change the values of persistent variables and to set up the process to handle the events that will cause its next transition. Special operations are used for this:

- *wait(e, s)* puts the process into the waiting state. When the event e occurs, the process changes its state to ready, and then whenever the CPU is allocated to the process it can go to state s.
- *delay(ms, s)* is similar, but the event being waited for is the timer expiring.
- *trigger(e)* causes event e to be signaled.
- *goto(s)* causes the process to directly jump to state s.

**Events** are the communication channels between processes. Events that are defined at the beginning of the node specification can be triggered by any process running in an active state. Timer events are triggered behind the scenes on the expiration of a timer. There is a timer for each process that can trigger the time event for just that process, without other processes being notified.

### 4.2.2 Type System

Our DSL has base types from NuSMV, such as **int** for integers and **boolean** for logical values. In order to facilitate the definition of a sequence of variables, our DSL also supports arrays of bits to be used for programming bit strings. Methods have a method type, which indicates the number and types of its input arguments (a.k.a. parameters) and its return type. Therefore, the methods can be distinguished by only using their names as their signature. On method invocation, the number and type of the input arguments must match the number of type of the parameters in the definition of the method.

### 4.2.3 Operational Interpretation of a DSL program

A program written in our DSL needs to be translated into an actual Arduino program in order to execute.

Each process has a start state, which is the first state that is defined in the process body. When execution of the application starts, all the processes are ready to be activated at their start states. The scheduler iterates over processes and decides which single process to activate next. An activated process performs all the statements specified in the state body, sets the values for variables, sets the events that the process should wait for, and/or triggers some events. It then goes to the wait state and releases the processor.

Processes are always listening to the event triggers, so that whenever an event is triggered by a process, all the ones that are waiting for that event change their state from wait to ready. Whenever a process becomes ready it no longer cares about new events, and simply waits for its CPU turn to be activated.

| | |
|---|---|
| { } | groups a set of lines |
| [ ] | indicates an array type of boolean or integer |
| ( ) | (1) indicates a higher priority expression, (2) shows a list of method arguments, or (3) includes number of bits an int variable has. |
| $/* */$ | indicates a comment |
| ; | end of the line |
| , | separates method parameters |
| # | indicates the include lines |
| $==$ | is equal to, |
| $!=$ | is not equal to |
| $+, -, *, /$ | binary mathematical operators for numerical expressions |
| &&, \|\| | binary logical operators AND, OR |
| ! | unary logical operator NOT |

Table 4.1: ArduinOS DSL Punctuation and operators

### 4.2.4 Lexical Elements

The basic tokens of the DSL are the following.

**Keywords** The keywords of this language are: node, event, input, output, process, root, state, trigger, wait, delay, goto, return, int, boolean, if, else, while, true, false.

**Identifiers** provide names for program elements: node, process, state, event, method, or variable. An identifier is a case sensitive word that starts with an underscore or letter and can be followed by any number of letters, underscores and digits. Identifiers cannot collide with keywords.

**Literals** are constant values of integer or Boolean type. An integer literal is either 0 or a sequence of digits 0 to 9 not starting with 0; a boolean literal is true or false. Each bit in bit strings can also be 0 or 1 like a regular boolean variable.

**Punctuation and Operators** are used to form statements and expressions. Operators manipulate integers and booleans. Table 4.1 shows the list of punctuation and operators that may occur in a program written our DSL language.

**White Space and Comments** are the usual lexical elements, with the usual non-nesting of /* */ comments.

### 4.2.5 Grammar

Here is the grammar of the DSL expressed in the usual BNF form:

– $\longrightarrow$ means the left hand side non-terminal can be replaced by the right hand side set.

– Non-terminals are printed in bold typewriter font. e.g., **Expression**.

– Terminals are shown inside “”, e.g., “node”.

– X ? means zero or one occurrence of X.

– X $*$ means zero, one or more occurrences of X.

– | indicates alternatives.

– ( ) groups multiple syntactical elements.

| | | |
|---|---|---|
| **NodeDefinition** | $\longrightarrow$ | (**IncludeHeader**)∗ “node” **Identifier** “{” |
| | | (**EventDeclaration** \| **I/ODeclaration** \| **VarDeclaration**)∗ |
| | | (**ProcessDeclaration**)∗ **RootProcessDeclaration** (**MethodDeclaration**∗ “}” |
| **IncludeHeader** | $\longrightarrow$ | “#” “include” Identifier “;” |
| **EventDeclaration** | $\longrightarrow$ | “event” **Identifier** “;” |
| **I/ODeclaration** | $\longrightarrow$ | (“input” \| “output”) **Identifier** (“[” **Expression**“]”)¿‘;” |
| **VarDeclaration** | $\longrightarrow$ | **Type Identifier** “;” |
| **ProcessDeclaration** | $\longrightarrow$ | “process” **Identifier** “{” (**VarDeclaration**)∗ (**StateDeclaration**)∗ “}” |
| **RootProcessDeclaration** | $\longrightarrow$ | “process” “root” “{” “state” “ROOT_INIT” “{” (“process” **Identifier**“;”)∗“}” “}” |
| **MethodDeclaration** | $\longrightarrow$ | **Type Identifier** “(” ( **Type Identifier** ( “,” **Type Identifier** )∗ )? “)” |
| | | “{” (**VarDeclaration**)∗ (**Statement**)∗ “return” **Expression** “;” “}” |
| **StateDeclaration** | $\longrightarrow$ | “state” **Identifier** “{” (**VarDeclaration**)∗ (**Statement**)∗ “}” |
| **Type** | $\longrightarrow$ | “int” “[” **Expression**“]” |
| | | \| “boolean” “[” **Expression**“]” |
| | | \| “boolean” |
| | | \| “int” |
| | | \| **Identifier** |
| **Statement** | $\longrightarrow$ | “{” ( **Statement** )* “}” |
| | | \| “if” “(” **Expression** “)” **Statement** “else” **Statement** |
| | | \| “if” “(” **Expression** “)” **Statement** |
| | | \| “while” “(” **Expression** “)” **Statement** |
| | | \| **Identifier** “=” **Expression** “;” |
| | | \| **Identifier** “[” **Expression** “]” “=” **Expression** “;” |
| | | \| “trigger” “(” **Identifier** “)” |
| | | \| “wait” “(” **Identifier** “,” **Identifier** “)” |
| | | \| “delay” “(” **Expression** “,” **Identifier** “)” |
| | | \| “goto” “(” **Identifier** “)” |
| **Expression** | $\longrightarrow$ | **Expression** ( “+” \| “−” \| “/” \| “∗” ) **Expression** |
| | | \| **Expression** ( “&&” \| “\|\|” \| “<” \| “>” \| “==” \| “! =” ) **Expression** |
| | | \| **Expression** “[” **Expression** “]” |
| | | \| $\langle INTEGER\_LITERAL \rangle$ |
| | | \| “true” |
| | | \| “false” |
| | | \| **Identifier** |
| | | \| “!” **Expression** |
| | | \| “(” **Expression** “)” |
| **Identifier** | $\longrightarrow$ | $\langle IDENTIFIER \rangle$ |

Table 4.2: ArduinOS DSL Grammar

### 4.2.6 Program Construction Constraints

The usual rules, such as about type consistencies in statements and expressions, are the same as the ones used in C/C++ languages. There are some additional domain specific rules, regarding the specification of nodes, processes, states, and also the constraints in scope and usage and declaration of variables and methods. In this section we provide these domain specific semantic rules while organizing them in cohesive groups.

**Include files**

– In order to reuse the code written in other node specifications, the related files can be included at the beginning of a node specification. Accordingly, all the node elements defined for the included source files will be assumed to be defined also in the destination node specification. For example, if B.node includes A.node, it is recognized such that all the elements defined in A are copied in B. Besides, all semantic rules regarding restrictions on naming the elements will be applied on resulting B, e.g., no two elements can have the same name.

**Process Declaration**

– Processes must have distinct names.
– If the node has no processes, it does nothing.

**State Declaration**

– States of a process must have distinct names.
– If the process has no states, it does nothing.
– A single transition consists of executing the statements included in the state block, including the ones after a wait or delay method.

**Built-in Functions**

– All the event arguments used in methods trigger and wait must be defined in the node.
– All the states used in built-in methods delay and wait must be defined in the process.
– Invoking trigger(e) means the flag for event e is raised and is recognizable by all the processes that are waiting for that event.
– Invoking wait(e, s) means event e is added to the list of events the process is waiting for, and if that event occurs, the process becomes ready to be woken up at state s.
– Invoking delay(ms, s) means the built-in time event is added to the list of events the process is waiting for, after ms milliseconds the time event would be triggered, the process becomes ready to be woken up at state s.
– If the process recognizes one of the events it is waiting for, it goes to ready state and does not care about the occurrence of other events.

**Event Declaration and Usage**

– All the events being triggered or waited for by processes, must be defined at the beginning of node specification.
– Events must have distinct names.
– Timer events are pre-defined and implicit in the node specification.

**Variable Scope**

– Variables can only be defined at the beginning of the related scope, e.g., in all of the node variables should be defined after events.
– Variables defined in the node can be seen by all processes (and their states) and methods.
– Variables that are accessible in each scope must have distinct names.
– Variables are read-only in methods but can accept values in the body of processes' states.

**Method Declaration and Usage**

– Methods can only be declared at the end of the node specification.
– Methods declared in the node can be seen by all processes (and their states) and methods.
– Methods must have distinct names.
– Return type of the methods can only be int or boolean. So, int array and Boolean arrays cannot be returned as a result value of a method. In other words, we allow the methods just to return a basic type as their resulting value.
– Method overloading, several methods with the same name but different parameters list or different return types, is not allowed.
– Method invocation is not allowed inside a method. In other words, in the body of a method, we can just use the inline statements and cannot invoke any method to use their functionalities. This also means that our DSL does not support recursive methods.
– Methods can have the access to the variables defined in the node to read them, but not for assigning values to them.
– There is just one return expression right at the end of the method declaration.
– The compiler sees the methods as inline statement, a.k.a. macros. Therefore, it is like the body of the method is moved to the place where it is invoked.

We now describe the step-by-step translations that occur from a DSL program to NuSMV and to executable Arduino code. The main difference in process is caused by the granularity of a C program compared to a NuSMV model. they are both based on state machines, but the source program execution is procedural while NuSMV model updates all its variables at the same time and has no notion of sequential execution of the lines of code. On the other hand, the state model notion and the event triggering and process notification are not directly defined in Arduino, but the sequential execution is the same for both languages.

## 4.3 Translating into NuSMV Models

Much of our experience in the previous PicOS modeling exercise can be reused. For each node program the NuSMV model must contain the node pcb, scheduler, shareVariables and all processes defined in the program. Therefore, while modeling ArduinOS applications we use almost the same modeling guidelines provided in Appendix A. However, since the PicOS modeling guidelines are scattered and hard to follow, we presented an easy to follow outline of the general process of extracting the NuSMV model from the ArduinOS program, see Appendix B. Besides, the ArduinOS modeling procedures contains slightly changed based on what is included in ArduinOS DSL:

- In ArduinOS DSL we are allowed to use "if" and "while" statements, whereas in PicOS we did not include those in our modeling guidelines. These statements affect on the control flow of the program and cannot be translated to only a single-step variable assignment rule. For instance, the body of a while loop may executed more than once and therefore the translator should simulate these steps and possible control flows in the NuSMV model. Accordingly, a process may stay in a running state for more than one step, whereas in PicOS modeling we assumed that each running state is executed in a single step.
- In ArduinOS we assumed that the inter-node communications are possible through uni-directional. Therefore, the interrupt-based communications are not included in ArduinOS modeling. Instead, for each connection between a pair of nodes (u,v), two uni-directional channels are required, and accordingly each node needs a pair of input-output pins for each of its connections to other nodes.

At last, note that the translation process is provided in Appendix B, however, it excludes network element modeling. The reason is that The network element has no equivalent in the node program and its logic resides in the connections between nodes. In other words, the information about the network of the nodes is actually provided by wiring up the Arduino nodes together. The hardware connections need to be specified and be modeled separately, this aspect will be discussed in Section 4.5.

Listing 4.4 presents the generated code for the ArduinOS example provided in Listing 4.2. The NuSMV model also includes two nodes connecting through a bidirectional channel[2]. The goal of presenting the NuSMV model of this application is to provide an example of modeling network graphs and representing how input and output ports should be modeled in each Node element in the model.

```

------------------Network------------------
MODULE main
VAR
```

[2]we did not provide the NuSMV representation of the second node since it was mostly similar to the first one.

```
  interrupt : word[1];

  N0_o_led :  {0, 1};
  N0_o_chan_0 :  {0, 1};
  N0_i_chan_0 :  {0,1};

  N1_o_led:  {0, 1};
  N1_o_chan_0 :  {0, 1};
  N1_i_chan_0 :  {0,1};

  N0 : node_N0(N0_i_chan_0, N0_o_chan_0, N0_o_led);
  N1 : node_N1(N1_i_chan_0, N1_o_chan_0, N1_o_led);

ASSIGN
  interrupt := 0b1_0;
  N0_i_chan_0 :=  N1_o_chan_0;
  N1_i_chan_0 :=  N0_o_chan_0;

------------------node_N0------------------
MODULE node_N0(interrupt, i_chan_0, o_chan_0, o_led)
VAR
  currentId : 0..3;
  event : word[2];
  pcb : node_N0_PCB(currentId, event, i_chan_0, o_chan_0, o_led);
  shared : node_N0_SharedVariable(pcb, event, interrupt, i_chan_0,
      o_chan_0, o_led);
  scheduler : node_N0_Scheduler(currentId , pcb);

------------------PCB------------------
MODULE node_N0_PCB(currentId, event, i_chan_0, o_chan_0, o_led)
DEFINE
  lock := p_init.lock | p_red.lock;
VAR
  p_init : node_N0_process_p_init(0, currentId, event, i_chan_0,
     o_chan_0, o_led);
  p_red: node_N0_process_p_red(1, currentId, event, i_chan_0,
     o_chan_0, o_led);

------------------SharedVariable------------------
MODULE node_N0_SharedVariable(pcb, event, interrupt,i_chan_0,
   o_chan_0, o_led)
VAR
  internal: word[1];

ASSIGN
  init(internal) := 0b1_0;   -- no event is issued in the
     application

  event := interrupt :: internal;

  init(o_chan_0) := 0;
  init(o_led):= 0;

next(o_chan_0) :=
  case
    pcb.p_red.state = OFF: 1;
    pcb.p_red.state = WAIT_FOR_INPUT & i_chan_0 =1: 0;
    TRUE : o_chan_0;
  esac;

next(o_led) :=
  case
    pcb.p_red.state = ON: 1;
    pcb.p_red.state = OFF: 0;
    TRUE : o_led;
  esac;

------------------Scheduler------------------
MODULE node_N0_Scheduler(currentId , pcb)
ASSIGN
  init(currentId) := 0;
  next(currentId) :=
  case
    next(pcb.lock) : currentId;
    TRUE : (currentId + 1) mod 2;  --only two nodes are in pcb
  esac;

------------------p_init------------------

MODULE node_N0_process_p_init(pid, currentId, event, i_chan_0,
   o_chan_0, o_led)
DEFINE
  lock := state != wait & state != ready;
VAR
  pevent : word[2];
  pstate : array 0..2 of {s_init};
  state : {s_init, wait, ready};
  nextstate : {s_init};
  timer : 0..10;
  tstate : {s_init};

ASSIGN
  init(state) := ready;
  init(pevent) := 0b2_0;
  init(timer) := 0;
  next(timer) := 0;
  next(pevent) := 0b2_0;
  next(state) :=
  case
    state = wait & ((event & pevent) != 0b2_0 | timer = 1) : ready
       ;
    state = ready & pid = currentId : nextstate;
    state = s_init : wait;
    TRUE : state;
  esac;

------------------p_red------------------
```

```
MODULE node_N0_process_p_logic(pid, currentId, event, orientation
    ,i_chan_0, o_chan_0, o_led)
DEFINE
  lock := state != wait & state != ready;
VAR
  pevent : word[2];
  pstate : array 0..2 of {START, ON, OFF, WAIT_FOR_INPUT};
  state : {START, ON, OFF, WAIT_FOR_INPUT, wait, ready};
  nextstate : {START, ON, OFF, WAIT_FOR_INPUT};
  timer : 0..1024;
  tstate : {START, ON, OFF, WAIT_FOR_INPUT};

ASSIGN
  init(state) := ready;
  init(nextstate) := decide;
  init(pevent) := 0b2_0;
  init(timer) := 0;

  next(timer) := ;
  case
    state = wait & ((event & pevent) != 0b2_0 | timer = 1) : 0;
    timer != 0 : timer - 1;
    state = ON : 1024;
    state = OFF: 1024;
    state = WAIT_FOR_INPUT & i_chan_0=0 : 1024;
    state = WAIT_FOR_INPUT & i_chan_0=1 : 1024;
    TRUE : timer;
  esac;

  next(tstate) :=
   case
    state = ON : OFF;
    state = OFF: WAIT_FOR_INPUT;
    state = WAIT_FOR_INPUT & i_chan_0=0 : WAIT_FOR_INPUT;
    state = WAIT_FOR_INPUT & i_chan_0=1 : ON;
    TRUE : tstate;
  esac;

  next(pevent) :=0b2_0;
  next(nextstate) :=
  case
    state = wait & (event & pevent & 0b2_01) != 0b2_0 : pstate[1];
    state = wait & (event & pevent & 0b2_10) != 0b2_0 : pstate[2];
    state = wait & timer = 1 : tstate;
    TRUE : nextstate;
   esac;

  next(state) :=
  case
    state = wait & ((event & pevent) != 0b2_0 | timer = 1) : ready
       ;
    state = ready & pid = currentId : nextstate;
    state = START : ON;
    state = ON : wait;
    state = OFF : wait;
    state = WAIT_FOR_INPUT : wait;
    TRUE : state;
  esac;

-- Similar text for Node 2 omitted for brevity ...
```

Listing 4.4: NuSMV Model For 2-Node 1-Process Blinker

## 4.4 Translating into Aurduino Code

Translating a program written in the DSL to an executable Arduino program is straightforward. Since the code for a state transition is essentially C code, all that is really required is to provide a library of functions that provide the functionality of the scheduler and event manager, and implementations for the primitives like wait(), trigger(), etc. This is not particularly challenging, and the only thing that should be considered while translating the ArduinOS into executable Arduino is to include the PicOS-based underlying structure for performing kernel behaviors. For instance, when an event is triggered all processes waiting for the event should be able to recognize it and update their status accordingly. We omit the translations details from here and explain them in Appendix C.

Listing 4.5 presents the generated code for the ArduinOS example provided in Listing 4.2.

```
template <class T, int N>

class myVector {
  T elements [N];
  int capacity;
  int currentIndex;

public:
  myVector (){
    capacity = N;
    currentIndex =0;
  }
  void add(T value) {
    elements[currentIndex] = value;
    currentIndex++;
  }
  void clear(){
    for (int i=0; i<currentIndex ; i++) {
      elements[i] = NULL;
    }
    currentIndex =0;
  }
  int size(){
    return currentIndex;
  }

  T get(int x) {
    return elements[x];
  }
```

```
};


class Process {
public:
  int timer ;
  int timeState;
  myVector <int, MAX_EVENTS> waitingStates;
  myVector <int, MAX_EVENTS> waitingEvents;
  int nextState;
  int state;
  virtual void run(int &i_chan_0, int &o_chan_0, int &o_led) =0;

  Process(){
    timer =0 ;
    state =ready;
  }
};


class P_INIT : public Process {
public:
  enum activeState { s_init };
  activeState activestate;

  P_INIT()
  : Process(){
    nextState = s_init;
  }

  void run(int &i_chan_0, int &o_chan_0, int &o_led){
    if (state==ready) {
      activestate = (activeState) nextState;
      state = active;
    }
    else return;

    switch (activestate) {
      case s_init:
        o_led =0;
        o_chan_0=0;
        break;
      default :
        ;
    }
    state = wait;
  }

};

class RED : public Process {
public:
  enum activeState { START, ON, OFF, WAIT_FOR_INPUT };
  activeState activestate;
  RED()
  : Process(){
    nextState = START;
  }

  void run(int &i_chan_0, int &o_chan_0, int &o_led){
    if (state==ready) {
      activestate = (activeState) nextState;
      state = active;
    }
    else return;

  decideCase:
    switch (activestate) {
      case START:
        activestate = ON;
        goto decideCase;
        break;
      case ON:
        o_led = 1;
        timer=1024;
        timeState=OFF;
        break;
      case OFF:
        o_red = 0;
        o_chan_0 =1;
        timer=1024;
        timeState=WAIT_FOR_INPUT;
        break;
      case WAIT_FOR_INPUT:
        if (i_chan_0 ==0){
          timer=1024;
          timeState=WAIT_FOR_INPUT);
        }
        else{
          o_chan_0 =0;
          timer=1024;
          timeState=ON;
        }
        break;
      default :
        ;
    }
    state = wait;
  }

};

myVector <Process*,MAX_PROCESSES> pcb;

void trigger (int event){
  for (int i=0;i<pcb.size();i++){
    for (int e=0;e<pcb.get(i)->waitingEvents.size();e++){
      if(pcb.get(i)->waitingEvents.get(e) == event){
        pcb.get(i)->nextState = pcb.get(i)->waitingStates.get(e);
        pcb.get(i)->waitingEvents.clear();
        pcb.get(i)->waitingStates.clear();
        pcb.get(i)->timer =0;
        pcb.get(i)->state = ready;
      }
    }
  }
}

void globalTimertick(){
  delay(100); // added to make the behaviour observable
  for (int i=0;i<pcb.size();i++){
    if(pcb.get(i)->timer != 0){
      pcb.get(i)->timer = pcb.get(i)->timer - 1;
      if(pcb.get(i)->timer == 0){
        pcb.get(i)->nextState = pcb.get(i)->timeState;
        pcb.get(i)->waitingEvents.clear();
        pcb.get(i)->waitingStates.clear();
        pcb.get(i)->state = ready;
      }
    }
  }
}

void setup(){
  pinMode(OUTPUT_PIN0, OUTPUT);
  pinMode(OUTPUT_PIN1, OUTPUT);

  pinMode(INPUT_PIN0, INPUT);

  //the PCB element
  pcb.add(new p_init());
  pcb.add(new RED());
}

void loop(){
  for (int i=0;i<pcb.size();i++){
    //read from inputs
    i_chan_0 = digitalRead(INPUT_PIN0);

    pcb.get(i)->run(i_chan_0, o_chan_0, o_led);

    //update the outputs
    digitalWrite(OUTPUT_PIN0, o_led);
    digitalWrite(OUTPUT_PIN1, o_chan_0);

    globalTimertick();    // ticks every process
  }
}
```
Listing 4.5: C++ Code For Arduino Executable (Node 1)

## 4.5 Covering the gap between real world application and ArduinOS NuSMV model

Every program written in our DSL includes the logic for a single node behavior. This is used to generate the code for a single node. On the other hand, the NuSMV model needs to contain the model for all the nodes in the network. It needs to know how the nodes are connected to each other, if the nodes move in the network, and etc. so that it can check if the complete distributed application can satisfy requirements on that specific topology. Therefore, all network related information that cannot be assumed locally by each node must be added to the NuSMV model. Following are some issues that should be considered before performing the model checking procedure on the NuSMV model.

**Network topology**

Information about the network topology is neither programmed in the DSL program nor in anywhere else. This should directly be resulted from the deployment of application on a set of nodes, and how the nodes are connected. Therefore, in order to have a complete NuSMV of the network application, we need to somehow feed the information to the model. A user familiar with NuSMV modeling can simply modify the main module of the model and build the network graph model. However, in order to have a more user friendly interface, the DSL could be extended to define the structure of the network and saying what application should be deployed on each node, and therefore the translator should be able to integrate those information into the target model. In either way, the followings should be presented to the NuSMV model. Assuming the network as a graph $G(V, E)$ where $V$ is the list of nodes and $E$ in the list of channels:

- For all $v \in V$, v should be initialized. The list of nodes in the network, and their corresponding node definitions (.node file) should be set. The application uploaded on each target node should be known while initializing the node in the NuSMV model.
- For all $e = (v, u) \in E$ a pair of input and output channels are needed for each of the nodes, so that v_input = u_output and u_input = v_output.
- Depending on the applications defined for each node, if the node needs to know about its input or output degree in the network, this information should be passed to the node while initialization. The degree of the node in the network is directly resulted from the structure of the network and cannot be locally realized by each node, unless it is passed to the node from the beginning.

**Unique identifier**

In some algorithms, each node in the network needs to have a unique identifier. Since each node does not have any notion about the uniqueness of the ID, it is considered as network-layer information and should be set while initializing the network. Therefore, while initiating the nodes in the network, for nodes requiring an ID, a unique number should be passed to the node.

**Dynamic Network Topology**

In real networks, the network topology may change over time. For instance, a node runs out of battery and is deleted from the network, some links get disconnected over time, and then reconnect.

In ArduinOS framework, we can also include topology changes to some extent. We assumed that even before the execution we know how the topology is going to change and we know exactly when this change is happening. In that case, we can extend our NuSMV model so that we can check our algorithms for dynamic topologies too.

Dynamism of network is a time-based property of the application. In order to model such a property, a counter must be modeled so that it increases at each step, and sequences the change in topology.

In order to model the time-based topology changes in NuSMV, the following steps should be done while defining the network. In this regard, we assume the initial network topology is the graph $G(V, E)$, and it wants to change to $G'(V', E')$ at $i$th step:

- For all $v \in (V \cup V')$, v should be initialized.
- For all $e = (v, u) \in (E \cup E')$ the link should be modeled as follows:
  - If $e \in E$ and $e \in E'$, we have v_input = u_output ( and u_input = v_output ), and we call it a regular link.
  - If $e \in E$ and $e \notin E'$, we should have a conditional assignment for the inputs so that if $counte < i$ v_input = u_output unless v_input = null (and so for u_input channel). We call such link a deleted link.
  - If $e \notin E$ and $e \in E'$, we should have a conditional assignment for the inputs so that if $counte < i$ v_input = null unless v_input = u_output (and so for u_input channel). We call such link an added link.
- If the input or output degree is needed to be passed to nodes, we can simply consider degree of the node in the graph $G'(V \cup V', E \cup E')$.

## 4.6 ArduinOS Framework Model

Figure 4.1 shows the conceptual model of the framework. The higher a layer is, the closer it is to the communication interface with the user:

- The programming interface can be any text editor for creating the node definitions (.node files).

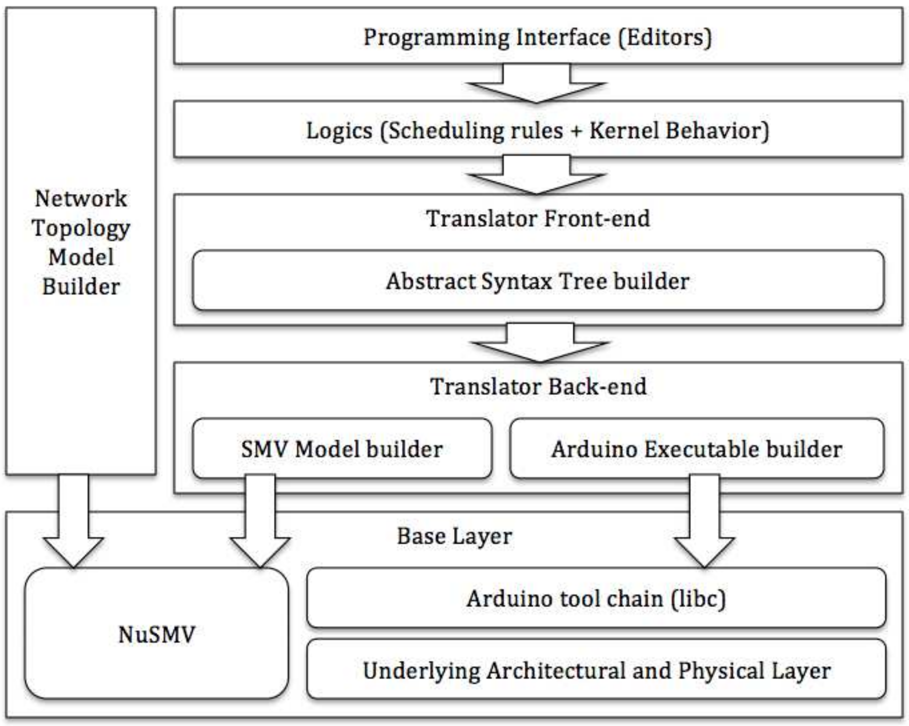
Programming Interface (Editors)
Logics (Scheduling rules + Kernel Behavior)
Network Topology Model Builder
Translator Front-end
Abstract Syntax Tree builder
Translator Back-end
SMV Model builder
Arduino Executable builder
Base Layer
NuSMV
Arduino tool chain (libc)
Underlying Architectural and Physical Layer

Figure 4.1: ArduinOS Framework Conceptual Architecture

- The networking topologies vary depending on the deployment of the application on different sets of nodes. The topology is set on hardware basis and the NuSMV model should respectively be modified. Accordingly, at the highest level, the user familiar with the NuSMV modeling should be capable of setting the network structure of the nodes in the model [3].

- The kernel logic layer, containing scheduling rules and the kernel level logic for coordinating the applications, resides under the direct user interface, and is logically placed at the top of the translator layer.

- The translator front end layer acts like an Abstract Syntax Tree builder in a compiler. This layer focuses on extracting element from DSL application structure, and finding the controlling procedures, such as loops and conditional statements, in application processes.

- The translator back end builds the target NuSMV and Arduino programs based on the extracted elements and their controlling logic. These functionalities are performed by two components in this layer NuSMV modeler and Arduino Executable builder. The former is responsible for building the NuSMV model (where the NuSMV model is modified by the user to contain the network topology) and presents it to the model checker, and the latter builds the executable program for Arduino devices.

[3] In order to facilitate this setting procedure for users we could also provide a graphical interface for managing the nodes and their links.

- The base layer (the lowest layer) consists of two individual components which are only used by the framework for model checking and behavior observing of the application and is not actually part of the proposed framework.

  - The NuSMV model checker for examining the behavior of the application. This component receives the NuSMV translation of the ArduinOS application and checks the correctness of some specifications. The specifications should be fed to the model checker based on the intended behavior and properties of each application.
  - The Arduino toolchain is used to compile and upload the Arduino executable program.

  We, of course, are making a big assumption that all the bits and pieces above are actually properly specified and implemented in ArduinOS applications.

# Chapter 5

# Experiments

In this chapter we provide an initial ArduinOS evaluation through a sequence of increasingly complex case studies:

1. Talking nodes, which illustrates the inter-node communication in a network to send and receive a one-bit token.
2. Self-stabilizing ring orientation, that obtains an agreement among the processors for a consistent notion of left and right.
3. Distributed clustering, that partitions that nodes of a network into clusters each of which having a leader (cluster head) and therefore provides a hierarchical organization for the network.

Validating the correctness of the framework for each of these experiments involves examining the ArduinOS to Arduino Executable translation, observing the actual execution of the application, examining the ArduinOS to NuSMV translation, and model-checking the resulted NuSMV model.

In the reminder of this chapter we explore each of these case studies and analyze them using our modeling framework. For each case, we introduce the problem, point out some assumptions about the modeled network, explain the algorithm, and then see if the framework can model the algorithm so that it satisfies the specifications.

## 5.1 Talking Blinkers

A fundamental feature of a network is the possible communication protocols between nodes. Every two adjacent nodes must be able to mutually send/receive messages to/from each other. In this section we address the problem of simple inter-node communication, also known as talking blinkers, where two nodes exchange messages over a link. Note that the communication channels in ArduinOS are different from the interrupt based ones used in PicOS modeling. Therefore, this case study can be considered as another test for the framework regarding the new communication protocol, and can be assumed as a complementary to Section 3.5.

### 5.1.1 Problem Definition

If two nodes can talk to each other, it means that they can mutually receive each other's messages, react to the received messages, and send messages back to the other node. Accordingly, other than the communication protocol, the behavior of each node and its reaction to the messages should work correctly. This case study, as the most basic one, aims at nodes talking and expressing the intended behavior based on their communication.

Since we want to demonstrate the fact that two nodes can send messages back and forth on a link the reaction of a node to a message should be modeled as an observable property. Accordingly, we decided to change the problem to two talking nodes which blink in turn. In order to have such behavior, other basic properties of the framework are required to work properly, for instance, event handling, process scheduling, state transitions, etc.

All things considered, the properties that should be satisfied by any solution to this problem are as follows:

- The nodes should blink one at a time, that is, changing the nodes outputs so that if connected to LEDs their states changes from off to on periodically.
- There should be no conflict in sending messages, so that at each time only one of the nodes is sending a message, and the other one is receiving.
- A message should be sent back and forth, so that once a node sends a message, it should wait till the other one receives it and sends it back.

Accordingly, here the main goal is to send a message (and only one message) back and forth between two nodes, and observe their behavior as blinkers (e.g., when a node received a message it blinks).

### 5.1.2 Talking Blinkers Assumptions For ArduinOS

This case study is actually an experiment for testing the correctness of ArduinOS fundamentals, and examining the framework performance based on PicOS-based assumptions. Therefore, the assumptions needed for designing the communication algorithm are mainly the ones considered while building the modeling framework. Other than the PicOS-based assumptions, we consider the followings while designing the communication protocol for ArduinOS.

- For the sake of modeling simplicity, each bidirectional channel is implemented using two single directional channel.
- If a channel is high, it means there is message transmitting on that channel.
- The output of a node is connected to the input port of the other one, and any changes in an output can be instantly recognized by the other one.

- There should be no message loss in the protocol.

### 5.1.3 Talking Blinkers Algorithm

In order to make the talking procedure complex enough to be interesting to model check, and also cover all functionalities needed for this program, we decided to modularize the design into three main processes as follows:

- p_poll_tick process, which is the driver of the whole system. It actually simulates the local clock of the node. It triggers an event, e_poll_tick, and then notifies the other processes that are waiting for that event.
- p_blink process, which is responsible for setting the outputs of the node, so that the behavior of a blinker could be observed. For this program, we decided to have a two-LED blinker, both being off at the beginning. Whenever the blink process is notified one of the LEDs turns on and the other turns off. Therefore, the p_blink needs to be a process having an initial state (off, off) and two main states (on, off) and (off, on), and it transmits between states by recognizing a blink event. In other words, at each state, the process is waiting for an event, say e_blink, and whenever the event is recognized it can change its state.
- p_tick process, which is activated after each e_poll_tick event is recognized and includes the main protocol for inter-node communication. Initially, one of the nodes is sending and the other one should start listening. The procedure running on both nodes are the same.
  - The procedure starts at Listening state. Node A checks if any input messages have been received (if the input channel is high). While the input is low, we wait for a clock tick (e_poll_tick) and continue listening. If it is high, it means that the other node (node B) is trying to send a message, so A's output should be low at that time and its state can change to Receiving state.
  - Entering the Receiving state, the node notifies an event (e_blink) to activate the blinker process, that will later result to a state transition in p_blink process. Then the node waits for a bit to make sure the message has been received thoroughly and then transmits to Sending state.
  - In Sending state, the node checks it if the input message has been completely received or not. If it figures out the other one is still sending the message, it keeps waiting till the input goes low. If the input is low, it starts sending its own message by setting high the output. Then the state of the node will be changed to CompleteSending.
  - In CompleteSending the node simply finishes the sending procedure by setting low the output and going back to Listening state, in which it wait to receive a message.

In summary, the node listens to the arriving message, receives the message (and blink) and waits till the transmission finishes, sends the message back to the other node and then starts listening again. Therefore, the network behavior should be: A sends a message, B receives it and blinks, then B sends a message, A receives it and blinks, and the message will be passed back and forth between nodes. This algorithm is similar to the example provided in Listing 4.1 in the sense that two nodes talk and blink their LEDs, but it is a little more advanced and has more possible states since we have two LEDs for each node and the resulted blinks are coordinated differently.

The mentioned communication protocol should be implemented in ArduinOS, and its corresponding Arduino executable and NuSMV model should be extracted by employing the proposed framework.

### 5.1.4 Verifying Talking Blinkers ArduinOS Application

The intended behavior of the communication protocol is straightforward, the two nodes in the network should blink their LEDs and change their status alternately. Knowing that the nodes react to received messages and change their LEDs' mode accordingly, we expect the two nodes executing this protocol will go through the cycle of modes:

(XX, XX), (RX, XX), (RX, RX), (XG, RX), (XG, XG), (RX, XG), (RX, RX), ...

where (XX, XX) demonstrates the ordered pair of (node1-LEDs, node2-LEDs); X means the LED is off, R means the red LED is on, and G shows that the green LED is on.

Accordingly, the program should start execution from the state where all LEDs are off. One of the nodes starts sending the message and the other one blinks its LEDs in response and change its status to RX (now the system status is RX for one node and XX for the other one). Then the received message should be sent back and result to LEDs blinking for the first node, and changing the system status to (RX, RX). The remainder of the execution is the same in the sense that the nodes receive messages and blink.

In order to verify the application, we check if the mentioned cycle of nodes is the only possible execution path. If not, one of the followings caused that: 1) the original algorithm is not programmed correctly in ArduinOS, which means the ArduinOS application should be fixed and then its corresponding NuSMV model should be checked against the specification; 2) the model is not representing the application properly, which means the framework has failed modeling the ArduinOS program and it should be revised. Among these possibilities, we are interested in the second one since validating the framework is what we focused on in this case study.

We used CTL to specify the execution cycle and checked the model on the specification. We need the following specifications in order to specify the execution path starting from (XX, XX) and eventually continuing in the above mentioned cyclic path [1].

[1] We do not present the CTL predicates for the specification since they can easily be interpreted from the explained specification rules explained in Chapter 2.1.3

- The first three states runs properly: the status of the LEDs (i.e. the corresponding output pins) are (XX, XX) at the beginning and they stay at that state until it goes to (RX,XX) and then keep that state until it changes to (RX, RX).
- The cycle is followed properly: if the status of system is (RX, RX) it should eventually changes to (XG, RX), (XG, XG), (RX, XG) in order and then changes back to (RX, RX).

The combination of the two above mentioned specifications guarantees the single intended execution path. The first part guarantees that the first three steps of the expected execution path are followed properly. It means that the program starts where all LEDs are off, and the first change it realizes is the first node's red light turning on, and then the next change would result to both nodes having only one red LED on. The second part guarantees the execution of cycle starting from (RX, RX) and finishing to the same state. It specifies that if we get to the state (RX, RX) we will continue the execution by changing states to (XG, RX), (XG, XG), (RX, XG) and (RX, RX) again. If the model verifies that, it means that the cycle is followed continuously or (RX, RX) can never be reached. However, according to the first part of the specification we know that the state (RX, RX) is reachable. Therefore, the execution of the program in the cycle would be proven.

## 5.2 Self-stabilizing Ring Orientation

Ring orientation is a well-known and studied problem in the context of distributed ring topology networks. A global orientation of a ring topology makes some problems easier. For instance, when a message needs to be transmitted through the whole network, the number of messages needed to be transmitted decreases if the network is oriented. Since the problem is crucial in the context of sensor network, and also because of its interesting observable behavior, in this section we address the orientation problem to implement in ArduinOS and correctness evaluation.

### 5.2.1 Problem Definition

In general terms, the orientation problem is to obtain an oriented network regardless of the initial configuration of the nodes orientations. A network is call oriented if for every edge in the network, exactly one of the nodes at the ends of it is oriented toward that edge.

The orientation problem in a ring topology is simply defined as finding a unicycle orientation for the ring network. Assuming a network of N nodes in a ring topology network, there are precisely two ways to orient the ring: 1) all edges are left oriented which result to a clockwise ring orientation, or 2) all edges in the network are right oriented and result to a counter-clockwise network orientation.

In a ring topology of sensor networks, where the nodes have no global view of the topology and its orientation, the strategy to solve the orientation problem is tricky. The algorithm should be distributed and should be the same for all the nodes in the network. Therefore, the nodes should work independently and should decide to which side they must be oriented based on the received messages from their neighbors. Accordingly, any algorithm solving the orientation problem for a set of independently executable nodes should satisfy the following properties:

- The algorithm should be self-stabilizing, meaning that regardless of the start state, it should eventually enter and remain a set of well-defined stable states, called the target set.
- Regardless of the nodes' initial orientation in the ring topology, the ring should be able to orient clockwise (or counterclockwise), meaning that all nodes in the ring network should face left (or right)
- If the coin-flips in picking a random state for node states are unfair, there may be some nodes facing each other, and the algorithm does not progress because of specific unfair chain of randomly selected states.

### 5.2.2 Ring Orientation Assumptions For ArduinOS

Distributed ring orientation algorithms should perform based on the state of a node and its communication with other nodes. Therefore, all assumptions regarding the communication and token passing mentioned in the previous sections should also be considered for designing a ring orientation program. There are some additional orientation specific modeling assumptions that should be encountered while modeling the algorithms for ArduinOS.

- The algorithm running on all nodes should be the same, with no distinguishing property between nodes.
- The nodes should be connected in a ring topology, each node having two neighbors.
- The nodes may run asynchronously, and the token passing mechanism used for node communication should handle that.
- The initial orientation of each node (right or left) is arbitrarily chosen.
- The sequence of coin tosses for links may occur unfairly in the environment and make the algorithm stick in specific states. Therefore the deadlocks resulted from unfair coin tosses should be ignored while evaluating the algorithm.

### 5.2.3 Ring Orientation Algorithm

The algorithm used for network orientation is adapted from the algorithm proposed by Hoover et al. [20]. This is a uniform distributed algorithm in which each node starts in a random orientation status and operates independently from other nodes and decides about its orientation based on the inputs received from its input ports. In this section, a big picture of the algorithm will be described and its corresponding ArduinOS implementation will be presented.

| Next Mode | | Input | | |
|---|---|---|---|---|
| | | W | P | R |
| Current Mode | W | (W,R) | R | W |
| | P | P | W | W |
| | R | P | R | W |

Table 5.1: Token Passing Protocol

The idea of the algorithm is that, each node decides about its next state and its orientation based on its current state and what it received from other nodes. At the beginning, each node faces to its right or left [2], and starts waiting for a token (the initial state of the node is W). If the node realizes the node on the other side of the channel is also waiting for a token, it randomly changes its status to W or R in order to make some change to the status of the system and result in progress. This is one of the situations where the randomness, and the fairness of the environment, affects the progress of the application. If both nodes stays at W again and again the application would get stuck. Accordingly, each node waits till receives a token from the side it is facing to, then it turns to the other side and tries to pass the token to the node at the new orientation.

The token passing protocol employed in this algorithm is illustrated in Table 5.1 obtained from [20].

In order to implement this notion in a state-based program for ArduinOS, we assume separate states based on the node's orientation (left or right) and its token-passing status on that orientation (W, P or R). Therefore, six main states are needed for implementing the logic of the algorithm. In order to make the coordination easier, we decided to add another state "decide" which is responsible for deciding about the next state of the node based on its updated orientation and token-passing status. For example, if the node's orientation is "left" and it is waiting to receive a token from its left (its token-passing status is W), then after a tick the node is supposed to be notified in state Wleft (simply resulted from W state and the node orientation). Listing 5.1 presents the ArduinOS representation of this algorithm.

```
node N0 {

  int orientation ;

  input i_right;
  output o_right;

  input i_left;
  output o_left;

  process p_init {
    state s_init {
      orientation = random{left, right};
      o_left = W;
      o_right = W;
```

[2]The nodes are deployed in a ring topology and have one right channel and one left channel each.

```
    }
  }

  // process to generate a polling tick to cause inputs to be
      sampled
  process p_poll_tick {
    state s_0 {
      trigger(e_poll_tick);
      delay(1, s_0);
    }
  }

  process logic {
    state decide{
      if(orientation== left & o_left == W)
        wait(e_poll_tick, Wleft);
      if(orientation==right & o_right == W)
        wait(e_poll_tick, Wright);
      if(orientation==left & o_left == P)
        wait(e_poll_tick, Pleft);
      if(orientation==right & o_right == P)
        wait(e_poll_tick, Pright);
      if(orientation==left & o_left == R)
        wait(e_poll_tick, Rleft);
      if(orientation==right & o_right == R)
        wait(e_poll_tick, Rright);
    }
    state Wleft{
      if (i_left == W)  o_left=random(W,R);
      else if(i_left == P) o_left=R;
      goto(decide);
      }
    state Wright{
      if (i_right == W)  o_right=random(W,R);
      else if(i_right == P) o_right=R;
      goto(decide);
    }
    state Pleft{
      if (i_left == P)  o_left=W;
      else if(i_left == R) o_left=W;
      goto(decide);
      }
    state Pright{
      if (i_right == P)  o_right=W;
      else if(i_right == R) o_right=W;
      goto(decide);
    }
    state Rleft{
      if (i_left == W) {
        orientation = right;
        o_right=P;
      }
      else if(i_left == R) o_left=W;
      goto(decide);
      }
    state Rright{
      if (i_right == W){
        orientation = left;
        o_left=P;
      }
      else if(i_right == R) o_right=W;
      goto(decide);
      }
    }

  process root() {
    state ROOT_INIT {
      process p_init;
      process p_poll_tick;
      process logic;
    }
  }
}
```

Listing 5.1: ArduinOS Code For Token Passing

### 5.2.4 Verifying Ring Orientation ArduinOS Application

The intended behavior of the Ring Orientation protocol is to orient all the nodes in the network to one direction, either left or right. The target network for verification of this algorithm contains nodes in a ring shape, where for every node A and B in the network, A is in the right side of B iff B is on its left side. The number of nodes in the network can be two or more, but as the number of nodes in the network grows the time needed for the network to be stabilized and oriented increases. We base our experiment on the smallest network which could be interesting to check, a network of three nodes connected as a triangle.

In order to verify the correctness of the application, we applied an specification combining the following two behavior presented as follows. The specification satisfying both parts of the intended behavior guarantees that the application is self-stabilizing and will result to properly oriented network.

- Ignore the execution paths which get stuck because of unfair coin tosses.
- In a fair environment, the network can eventually be oriented regardless of its start state, i.e. eventually all nodes will face to right, or they will all face to left.

The first part guarantees that the execution is done in a fair environment. Note that if two nodes in the network are facing to each other and they are both W (or both R), they need to flip a coin to select their next value from W and R until they get different states (one become W and the other become R). Even if flipped fairly they might get the same values (both Ws or both Rs), but this lack of progress cannot continue indefinitely. The expected number of coin tosses for a pair of fair coins before they differ is 2. Therefore, we expect the coins to have different values after two tosses, unless the coins are not tossed fairly and we should ignore that state. Accordingly, if the history of coin tosses shows that the tossing results have been the same for both nodes we do not continue executing that path, and therefore, that state should not be considered for the orienting network specification check (see Section 2.1.4).

The second part assumes the environment is fair (obtained from the first part of the specification) and guarantees that regardless of start orientation of the nodes, they will be all agree on the orientation and eventually result an oriented network. If our NuSMV model satisfies this specification it should be a correct representation of the application and increases our confidence in the correctness of our proposed framework.

We checked the NuSMV model with the above mentioned specification, and it was verified for the triangle network, ring network of three nodes. It took the model checker about 34 hours to complete on this small network of 3 nodes. We also checked the application for a ring of four nodes but the time needed to check the specification increases so much that we stopped the execution.

## 5.3 Distributed Clustering

Obtaining a hierarchical organization of a network is a studied and well-known problem in the context of distributed networks. Clustering a network means partitioning its nodes into clusters, so that each cluster has one clusterhead (leader) and possibly some ordinary nodes. Clustering can be used for hierarchical routing and also for building and maintaining cluster-based network architectures.

### 5.3.1 Problem Definition

Sensor networks can be modeled as graph of nodes. Each node in the network is assigned a unique identifier (ID). A link between two nodes illustrates a bidirectional channel through which nodes can mutually receive the others' messages. Accordingly, if there is a link between nodes $v$ and $u$ we simply call them neighbors or adjacent nodes. The topology of a simple network is shown in Figure 5.1.

Clustering a network is simply partitioning its nodes into some groups. Each group (cluster) needs a clusterhead (leader) and some ordinary members. The choice of the clusterhead can be based on any weighted criteria. For the sake of simplicity, in this case study, we use the IDs of the node as the weighted criteria, so that the smaller the ID of the node, the better that node for the role of clusterhead. Therefore, in each neighborhood, the node with smaller ID can be a clusterhead, and its neighbors affiliate with it only if they have no other neighboring clusterhead with smaller ID. Figure 5.2 illustrates a correct clustering for the network of Figure 5.1. In this picture, the clusterheads are shown as squared nodes.

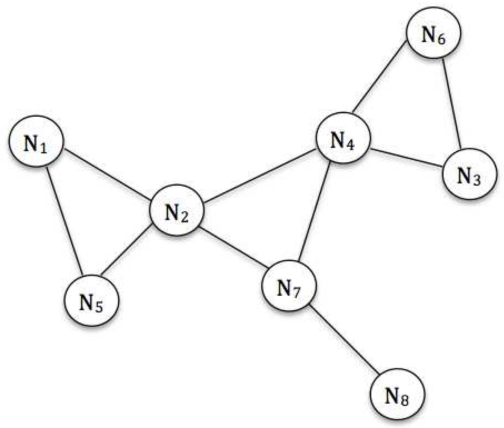

Figure 5.1: Example Network

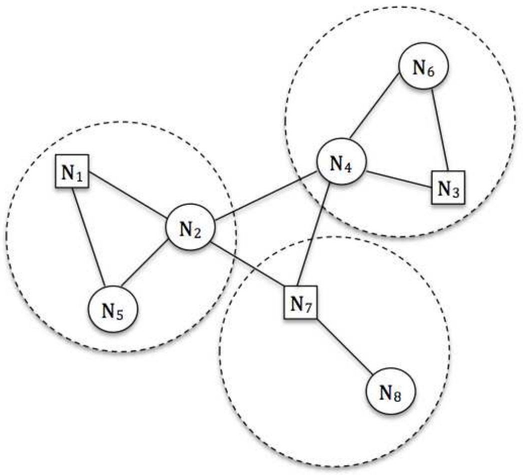

Figure 5.2: Clustered Network (Heads in Squares)

Figure 5.3: Distributed Clustering Example

In order to properly partition the network, the clustering algorithms should satisfy the following properties:

- Each node should be either a clusterhead or an ordinary node belonging to a cluster of a clusterhead is in its neighborhood.
- Each ordinary node has at least one clusterhead as neighbor.
- Each ordinary node affiliates with the neighboring clusterhead with the bigger weight (here smaller ID).
- No two clusterheads can be neighbors if their weights are unique.

The first property guarantees that all nodes in the network have been partitioned into clusters. The second property is necessary to ensure that each ordinary node has access to at least one clusterhead to affiliate with. The third property is actually another representation of the rule that the node with smaller ID will be opted as the clusterhead. Finally, the forth property guarantees that the network is covered by a well scattered set of clusterheads [6].

Using the unique IDs as the base of clusterhead selection, the last property will be automatically satisfied. If both of the two neighboring nodes are clusterheads, this means that they have equal IDs, but that is not true since IDs are unique. Therefore, the node with bigger ID could be an ordinary node for the other one, or it may affiliate to another clusterhead.

Before presenting the modeled clustering algorithm, and proving the other properties based on the model, we first point out the assumptions we made in designing the algorithm.

### 5.3.2 Clustering Assumptions For ArduinOS

Aiming at analysis of clustering algorithm, in this case study we mainly focus on implementing a simple model of a clustering. In this experiment, we try to build an Arduino environment that covers all necessary fundamentals for a clustering algorithm, is simple enough to be modeled, and also is feasible to be model checked. Accordingly, we make some assumptions and build the clustering model based on them. The modeling assumptions for clustering problem are as follows:

- The algorithm should be uniformly distributed, meaning that all nodes in the network are executing the same code.
- The communication between nodes are possible by sending and receiving messages through links.
- Each node has a local understanding of the topology, so that it only knows its ID and the number of input ports it has to receive messages, without knowing which node is in its neighborhood.
- There should be no message loss or conflicts in transmissions.
- Any transmission sent on a channel can be received by other side of the channel in the next step.
- The network topology, the way nodes are connected to each other, is fixed but the links can be activated or deactivated. It means that all the existing or future network connections should be set before running the algorithm, but they may be added or deleted to the network sometime during the algorithm execution.

All these considered, we designed a uniformly distributed algorithm that works for static topologies, where the number of nodes are fixed and the mobility of them can be modeled by deactivating some links and activating some others. More detail about the algorithm is described in the remainder of this section.

### 5.3.3 Clustering Algorithm

In this section we describe a distributed algorithm that given any network, sets up a clustering that satisfies the properties listed in previous sections. As it was mentioned previously, we assume that each node knows about its ID, and its connection ports, so that if something has been sent though the channel connected to that port, the node could receive it at the next step.

While designing the clustering algorithm for ArduinOS, we create the code for a single node. The assumptions about how the network links are handled is the concern of network layer modeling in NuSMV, which should be programmed manually based on what network topology is intended to be checked and how the links and nodes are making the graph of the network. The algorithm implementation for a simple network is provided in Listing 5.2.

The algorithm is executed at each node so that the node decides its own role (clusterhead or ordinary node) based on its own ID and the messages received from its neighbors. As a result of partitioning the network into clusters, each node in the network knows to which cluster it belongs. Therefore, each node should know the ID of the clusterhead with which it affiliates. Accordingly, the node of which ID is equal to the ID of the clusterhead is the leader of its cluster.

As it was mentioned previously, each node can be either a ClusterHead or an OrdinaryNode that knows to what the ID of its clusterhead is. In order to design the state based version of the clustering algorithm, we assume that ClusterHead and OrdinaryNode are the main states (roles) a node can have, and the state transitions can occur based on the received messages from neighbors.

Initially, each node assumes it has the smallest ID in the neighborhood and therefore it is the clusterhead of its group. Therefore, for all of the nodes in the network, the clusterhead variable is actually the ID of the node.

After initialization, at each step the node should check all of its input ports to see what messages the neighbors have sent and how it affect the role of the node. There are two types of messages used in this algorithm:

- CH(v), used by node v, to send out its ID, and make its neighbors aware that it is currently assuming itself as a ClusterHead, and its ID is v.

- linked(v) used by node v, to send out its ID, and make its neighbors aware that it is currently affiliates with one of its neighbors, so that if others assume it is their clusterhead they should change their role.

Organized by the role a node can have, we have two main procedures as follows:

- If the node is in ClusterHead state, at each step it should check all of its input ports to see if any CH message has been arrived from a node with smaller ID than its or not. Therefore, the node goes over all inputs, if there is any CH message arrived, it checks if the sender ID is smaller than its own ID. If yes, the node should change its role to OrdinaryNode and send out a linked(ID) message.

- If the node is in OrdinaryNode state, at each step it should check all of its input ports to see if the clusterhead with which it affiliated has joined to another cluster or not. Therefore, the node goes over all inputs, if there is any linked message arrived, it checks if the sender's ID is the same as the node's clusterheadID. If yes, it means that the node to which we have joined is not a clusterhead anymore and our role should be changed to ClusterHead, and a CH(ID) message should be sent out.

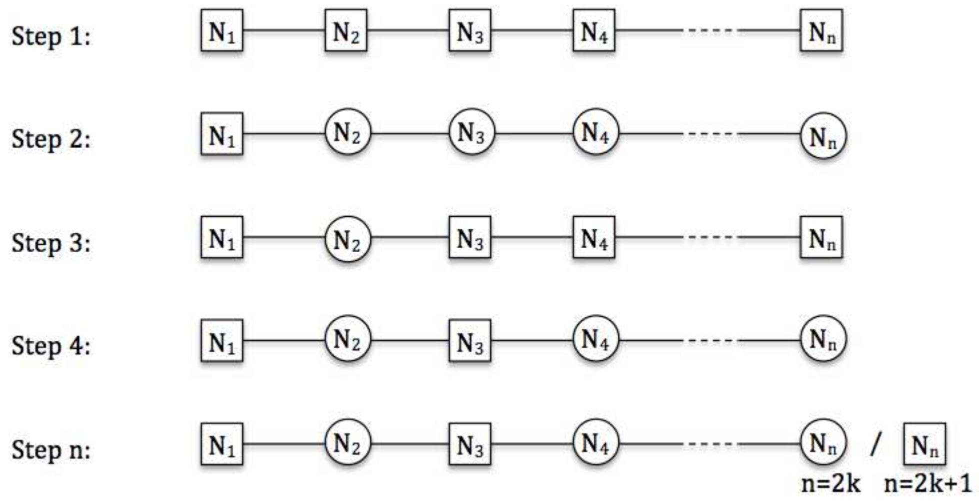

Figure 5.4: Worst Case Scenario of Clustering Algorithm (Heads in Squares)

The number of steps a fixed n-node network needs to be clustered and reach a stabilized state where all nodes have their final roles is O(n). The worst-case scenario of the algorithm is when we have a chain of nodes and the nodes are ordered ascending based on their IDs, Figure 5.4. In that case, all of the nodes assume themselves as ClusterHead, but in the next step except the first node, they all find a smaller node in the neighborhood and join it (they go to OrdinaryNode state), Figure 5.4. At that state, only the second smallest node is linked to the appropriate clusterHead, and the other ones has received the linked message from their assumed clusterhead, so they need to change back to ClusterHead role. The scenario goes on the same and at each step only one of the nodes gets to its final role. Therefore, the number of needed steps is O(n).

Regarding the number of state transitions needed and the complexity of the algorithm is $O(n^2)$ in the worst-case scenario. Assuming the same example shown in Figure 5.4, at each step the state of all of the nodes except one changes. That will result to $i$ state transmissions for the $i$th node, and a total of $\frac{n(n-1)}{2}$ state transitions.

Although the algorithm is not the most optimized clustering algorithm it guarantees partitioning the network into appropriate clusters [3]. We chose this algorithm since it needed only small messages to be transferred through channels and is also easier to implement on a Arduino based environments.

Listing 5.2 presents the ArduinOS representation of this algorithm.

```
node N0 {

  int clusterHead ;
  int degree; //this should be set based on the topology

  input i_msgType[degree]; //an array of size degree
  input i_msgContent[degree];
```

[3]For algorithms will less message transmission complexity see [6].

```
output o_msgType;
output o_msgContent;

process p_init {
  state s_init {
    clusterHead = myNodeID;
    o_msgType=CH;
    o_msgContent= myNodeID;
  }
}

// process to generate a polling tick to cause inputs to be
   sampled
process p_poll_tick {
  state s_0 {
    trigger(e_poll_tick);
    delay(1, s_0);
  }
}

process logic {
  int counter;
  state decide{
    if(clusterHead = myNodeID) {
      o_msgType=CH;
      o_msgContent= myNodeID;
      wait(e_poll_tick, CH);
    }
    else{
      o_msgType=linked;
      o_msgContent= myNodeID;
      wait(e_poll_tick, linked);
    }
  }

  state CH{
    counter = 0;
    while( counter < degree){
      if(i_msgType[counter] = CHmsg & i_msgContent[counter]<
         clusterHead){
        clusterHead = i_msgContent[counter];
      }
      counter = counter+1
    }
    goto(decide);
  }

  state linked{
    counter = 0;
    while( counter < degree){
      if(i_msgType[counter] = linkMsg & i_msgContent[counter]=
         clusterHead){
        clusterHead = myNodeID;
      }
```

```
        counter = counter+1
      }
      goto(decide);
    }
  }

  process root() {
    state ROOT_INIT {
      process p_init;
      process p_poll_tick;
      process logic;
    }
  }
}
```

Listing 5.2: ArduinOS Code For Clustering Algorithm

### 5.3.4 Verifying Clustering ArduinOS Application

The target network for verification of this algorithm can have any arbitrary topology, and the result of partitioning the network and the role of including nodes varies depending on what topology is used. Therefore, there is no specific rule to check the correctness of clustering, like what we did for verifying the Ring Orientation algorithm in Section 5.2. Furthermore, the execution path is so complicated and can be too long that is not feasible to be checked as we did for the talking blinkers in Section 5.1. Instead we know how the clusters will be stabilized eventually. Therefore, we can check if the expected final state is reachable and if the nodes keep that state.

For example, assume the example illustrated in Figure 5.3, in this example the final status of the nodes should be as follows ($N_X.clusterHead = Y$ means the ID of the head of the cluster to which node X is affiliated is Y. If X=Y it means X is the leader of its cluster.):

$$N_1.clusterhead = 1 \text{ \& } N_2.clusterhead = 1 \text{ \& } N_5.clusterhead = 1 \text{ \& }$$
$$N_3.clusterhead = 3 \text{ \& } N_4.clusterhead = 3 \text{ \& } N_6.clusterhead = 3 \text{ \& }$$
$$N_7.clusterhead = 7 \text{ \& } N_8.clusterhead = 7$$

We checked such specifications for multiple networks with various topologies and different number of nodes. Since the execution path does not contain any non-determinism branches, its model checking is performed in a reasonable time, much faster than our experiment in Section 5.2.

# Chapter 6

# Conclusion

## 6.1 Summary Of Research Results

This thesis covers the applicability of model checking techniques to the area of stateful processors aggregated into simple networks through low cost wired and wireless connection schemes. In this sense a development framework was proposed which enables both code generation and program verification in such networks. Based on the fundamental verifiable properties of applications in this area, a simple domain-specific programming environment was designed to generate both the model for performing verification via model checking and extract executable code that runs on the Arduino computing platform. Accordingly, this thesis is a step towards reasoning about networks of reactive systems.

Originally motivated by the problem of "how to build a trustworthy sensor network" we chose PicOS as a sensor network platform that provides a natural fit into model checking tools for specification and verification of its applications and has a small operating system for coordinating the applications. We utilized its state-based event-driven approach to extract a modeling schema for representing PicOS applications as NuSMV models, and checked them against intended specifications thereafter. However, the unknown underlying logic of the PicOS kernel on the one hand, and the logic residing in the physical and lower levels of PicOS devices on the other hand, made our NuSMV models not completely accurate with respect to the original systems.

Accordingly, we decided to build a platform from bottom up, and only include those properties that are already modeled according to our PicOS experiments. In other words, a model driven approach was applied that allowed no property to be included in the new platform unless it was satisfiable by the NuSMV model, and therefore the result could be a subset of PicOS with distilled functionalities and behavioral properties guaranteed to be correct by construction. We named it ArduinOS since it was adapted from PicOS and its applications could be deployed on Arduino devices.

We evaluated the correctness of our ArduinOS development framework on three increasingly complex case studies. The examples covered the basic communication protocol between two nodes, the ring orientation algorithm, and a distributed network clustering algorithm. Employing the proposed framework in verification of ArduinOS applications revealed that the performance of the system is a function of the size of the NuSMV model for the target application (e.g the number of bits employed in the NuSMV model in general), the topology of the network in which the applications should be deployed and model checked (e.g. what nodes are included and how they are connected), and the complexity of state transition logic and flows in the source program (e.g. the number of "if" and "while" statements used in the application). The results achieved from the experiments supported the correctness of ArduinOS framework both for developing the Arduino executable programs and generating an appropriate NuSMV abstraction for verifying its behavioral specifications.

Our approach toward verification of state-based systems applies to any distributed context in which nodes run independently from each other. Although our approach was motivated by verification of sensor network application, it can be applicable to verification of (distributed) systems and their programming frameworks in general, at least those networks of simple programmable nodes with small operating systems. However, it is not at all clear if this can be scaled to distributed applications deployed on more complex operating system. For instance, the same approach can be used in verification of some web based applications in which client nodes are communicating with each other and with a server, and can be coordinated based on server rules in servicing them, but only if they have the complexity of simple FSMs with limited messages.

We proposed ArduinOS framework based on PicOS-like systems so that ArduinOS applications are subsets of PicOS ones. In other words, every application that is programmable in ArduinOS can also be programmed in PicOS. We even designed ArduinOS DSL as a subset of the PicOS language; therefore, the applications in both frameworks are defined as a set of finite state machines. All these considered, ArduinOS is a simplified version of PicOS in which the functionalities are limited and the rules are more constraining. Consequently, the applications written in ArduinOS DSL can easily be translated into PicOS programs and deployed on PicOS nodes. However, it is worth noting that, even after translating the application from ArduinOS DSL to PicOS, in order to keep the verifiability of the application, we should force the simplified networking constraints of ArduinOS onto the PicOS system. For instance, the communications between the PicOS nodes should be limited to one-bit uni-directional channels and any other construction of packets should not be allowed. Therefore, digital ports in ArduinOS then transfer over to digital ports in PicOS, with the fact that the ports are implemented by packet messaging hidden from the application. Accordingly, ArduinOS framework and its verifiable applications also tell us about how a verifiable PicOS framework should be designed. It indicates what the minimal collection of properties included in PicOS should be.

On the other hand, examining a variety of ArduinOS applications and observing their behavioral outcomes, we recognized the shortcomings of some of the PicOS concepts and design decisions. Much of our model complexity resulted from transferring PicOS ideas that are seldom if ever used in practice over to ArduinOS. Therefore, ArduinOS framework could implicitly tell us how PicOS version 2 should go to fix its pitfalls.

- In ArduinOS we limited the root process to instantiate and start the other threads, and did not allow the processes to be instantiated in other places; whereas PicOS does not specify any limitations in this regard. Starting the threads all from root elemet not only makes the programs more structured and understandable, but also ensures that the PCB structure is constant (not dynamic) and makes the scheduling procedures more robust.

- Assuming state transitions to be atomic, means that there is no problem regarding the concurrency of threads since no shared variables would be modified by more than one thread at a time and only one thread can have the access to it at a time.

- Currently, it is possible that more than one process in a processor be waiting for the same event. If the event is triggered then all of them will be notified. However, it is also possible that the incoming event is actually used by only one of the processes. The others that are notified and then run (after waiting for being scheduled) realize that the event cannot provide any useful information for them at that specific time. In other words, only one of the processes is consuming and the others just wake up to nothing. There seem to be in practice very few cases of where more than one process actually needs to wait for a specific event. Accordingly, this property of PicOS may not actually be helpful in practice, and only one process should be the target of an event. PicOS can currently solve the problem of many processes waiting by defining specific events for each process, and thus provides a distinction between targeted events (to a specific process) and broadcast events (to all processes). Given the modeling and execution cost of broadcast events to multiple processes, and it low benefit, PicOS version 2 could eliminate this problem thoroughly.

## 6.2 Suggestions For Further Research

An obvious next step for this study is to perform a comprehensive evaluation of the development framework. We are reasonably convinced of the correctness of the modeling approach and ArduinOS framework was evaluated during the tests and case studies performed previously in this thesis. However, the cost of the modeling procedure and time required for model checking such applications is still too big. Some optimization can certainly be applied to extracting the NuSMV model. For instance, there is no code analysis performed in the way we handle “while” and “if” statements, thus causing more branching than necessary.

Another future activity is enhancing the framework to provide a user friendly interface that can take the topology of the network, and the type application running on each of its nodes, and inject the network NuSMV representation into the target model. This functionality is currently performed by hand by a NuSMV expert who knows about the network topology. It requires knowledge about the inner logic of each node in the network, and is more than can be expected for a regular ArduinOS programmer.

In developing ArduinOS we only included those properties that we could already model according to our PicOS experiment. However, we could re-examine the original assumptions and build ArduinOS independently to allow pluggable features, with one or more potentially easier to model domains.

# Bibliography

[1] Network simulator ns-2. http://nsnam.isi.edu/nsnam/.

[2] Ieee standard systemc language reference manual, 2005.

[3] *PicOS operating system version 3.3*. Olsonet Communications, September 2010.

[4] E. Akhmetshina, P. Gburzynski, and F. Vizeacoumar. Picos: A tiny operating system for extremely small embedded platforms. In *Las Vegas*, pages 116–122, 2003.

[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[6] Stefano Basagni. Distributed clustering for ad hoc networks. In *Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks*, ISPAN ’99, pages 310–, Washington, DC, USA, 1999. IEEE Computer Society.

[7] Ananda Basu, Marius Bozga, and Joseph Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, SEFM ’06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Simon Bliudze and Joseph Sifakis. A notion of glue expressiveness for component-based systems. In *Proceedings of the 19th international conference on Concurrency Theory*, CONCUR ’08, pages 508–522, Berlin, Heidelberg, 2008. Springer-Verlag.

[9] Nicholas M. Boers, Ioanis Nikolaidis, Pawel Gburzynski, and Wladek Olesinski. Picos & vneti - enabling real life layer-less wsn applications. In Marten van Sinderen, Octavian Postolache, and Csar Benavente-Peces, editors, *SENSORNETS*, pages 53–58. SciTePress, 2012.

[10] Michaël Cadilhac, Thomas Hérault, Richard Lassaigne, Sylvain Peyronnet, and Sébastien Tixeuil. Evaluating complex mac protocols for sensor networks with apmc. *Electron. Notes Theor. Comput. Sci.*, 185:33–46, July 2007.

[11] Alessandro Cimatti, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Verifying systemc: A software model checking approach. In Roderick Bloem and Natasha Sharygina, editors, *FMCAD*, pages 51–59. IEEE, 2010.

[12] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, UK, 1982. Springer-Verlag.

[13] Akim Demaille. Probabilistic verification of sensor networks. In *In Proc. 4th IEEE Int. Conf. on Comput. Sci., Research, Innovation and Vision for the Future (RIVF06*, pages 45–54. IEEE Computer Society, 2006.

[14] Wlodek Dobosiewicz and Pawel Gburzynski. Protocol design in smurph. In *State of the art in Performance Modeling and Simulation*, pages 255–274. Gordon and Breach, 1997.

[15] Jin Song Dong, Jing Sun, Jun Sun, Kenji Taguchi, and Xian Zhang. Specifying and verifying sensor networks: An experiment of formal methods. In *Proceedings of the 10th International Conference on Formal Methods and Software Engineering*, ICFEM ’08, pages 318–337, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] David Harel, Robby Lampert, Assaf Marron, and Gera Weiss. Model-checking behavioral programs. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT ’11, pages 279–288, New York, NY, USA, 2011. ACM.

[17] David Harel, Assaf Marron, and Gera Weiss. Programming coordinated behavior in java. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 250–274, Berlin, Heidelberg, 2010. Springer-Verlag.

[18] David Harel, Assaf Marron, and Gera Weiss. Behavioral programming. *Commun. ACM*, 55(7):90–100, July 2012.

[19] Thomas Hérault, Richard Lassaigne, Frédéric Magniette, and Sylvain Peyronnet. Approximate probabilistic model checking. In Bernhard Steffen and Giorgio Levi, editors, *Proceedings of the 5th International Workshop on Verification, Model Checking, and Abstract Interpretation (VMCAI'04)*, volume 2937 of *Lecture Notes in Computer Science*, pages 73–84. Springer-Verlag, January 2004.

[20] H.J̃ames Hoover and Piotr Rudnicki. Uniform self-stabilizing orientation of unicyclic networks under read/write atomicity. *Chicago Journal of Theoretical Computer Science*, 1996(5), December 1996.

[21] Philip Levis and David Gay. *TinyOS Programming*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[22] Philip Levis and Nelson Lee. *TOSSIM: A Simulator for TinyOS Networks*, September 2003.

[23] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 126–137, New York, NY, USA, 2003. ACM.

[24] Benny Shimony, Ioanis Nikolaidis, Pawel Gburzynski, and Eleni Stroulia. On coordination tools in the picos tuples system. In *Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications*, SESENA '11, pages 19–24, New York, NY, USA, 2011. ACM.

[25] Kazem Sohraby, Daniel Minoli, and Taieb Znati. *Wireless Sensor Networks: Technology, Protocols, and Applications*. Wiley-Interscience, 2007.

[26] Frank Werner. *Applied Formal Methods in Wireless Sensor Networks*. PhD thesis, University of Karlsruhe, 2009.

# Appendix A

# Guidelines For Extracting NuSMV Model Of A PicOS Application

In this section, we provide a modeling guideline that will be used for extracting a NuSMV model from a PicOs application. We introduce all the modeling elements, explained in Section 3.3, and provide their structure and including logics with a top down approach; starting from process that includes the major logic of the application layer, continuing with inner logic of the node layer that resides in node, scheduler and PCB elements, to the lowest level logics in network to manage inter node links and message passing.

Each subsection aims to explore an element by providing the following information:

1. Detailed design of the element; introducing the variables included in the element specification that are needed for performing the objectives, and the parameters that should be passed to the element while defining it. These properties are the ports to communicate with other elements and/or read from their variables [1].
2. Implementation guidelines including guidelines for updating the value of each variable based on the targeted PicOS kernel behavior and the user defined application.

## A.1 Process

Modeling processes is one of the most crucial and complex parts of the modeling schema, and it is also the part that is affected by our modeling assumptions the most. For example, considering time as a discrete variable or assuming that every state runs in one step may make our model different from what we expect from a PicOS application. Although being different from the real world is the inevitable result of every modeling procedure, providing an abstraction with the most possible similar behavior to its original is important for model checking. Accordingly, based on the assumptions, we designed the process units that facilitate model checking only those specification that are consistent with these assumptions.

[1] According to NuSMV language specification, whenever an element, say element A, needs to have the access to element B, A should include B or B should be passed to A as one of its module parameters while defining A.

Processes are designed as NuSMV modules, and not as NUSMV processes which are special constructions, now deprecated, for handling certain types of execution schedules in a NuSMV model. All NuSMV modules go one step forward at each NuSMV clock, meaning that the value of all variables may change at each step; whereas just one of NuSMV processes are activated at each step and only the variables defined in that process may change their values. Although we want to handle the processes scheduled to run one at a time, we do not want to limit the variables of the units to be changed only if the process is activated. For example, when a event is triggered, we want all the waiting processes to recognize the occurrence of the event and each process that has been waiting for that event should change its state from wait to ready. Therefore, we decided not to use NuSMV process types since it would not let us to perform event handling as desired. Using NuSMV modules, however, needs considering many possible conditions for assigning a value to a variable, based on if the related process is running or not, e.g. if it is the process' turn and it is ready, its state can change to one of the active states, specifically the state in which it should be woken up after realizing the occurred event.

**Detailed Design**

Modeling the behavior of a process is not possible without modeling the elements of a process. In the following, the process variables and parameters of which symbols are repeatedly used while explaining our model are introduced.

- **pid** is the process identifier and is a unique number. The value of pid is static, and it is set in the PCB that is responsible for instantiating the processes and is passed to the process as an input parameter.
- **currentId** is the identifier of the process which is scheduled to get the CPU at the time, i.e. the currently running process id. This variable is defined in the node and modified by the scheduler of the node. All instantiated processes need to receive currentId as an input parameter, i.e. currentId should be passed to all process instances while defining them in PCB. By comparing the value of currentId with pid, a process can understand if it is its turn to run or not.
- **event** is a shared variable between all process instances of a node that is passed to each process as an input parameter of the module. Event contains the information about the events (and interrupts) that are triggered by other processes of the same node (or other nodes). According to the limitation of NuSMV in operating on arrays, we decided to have events as a set of bits, a.k.a. word, of which size depends on the number of used events and interrupts in the application. When a bit of the word is 1 it means that event has been triggered.
- **state** is a process variable which shows the current state of the process. The domain of state variable should contain ready and wait (which are generic for all processes) and the active states (which are the defined states in the FSM of the PicOS application, such as s1, s2, etc.).

At each time the process can be in one of these states, i.e. state can have one of the values ready, wait, or an active state. If the process is one of the active states, we say that process is "running"

- **nextstate** is a process variable which shows the next active state in which the process should be notified when it gets the CPU. In other words, whenever the process takes the CPU, its state will be what nextstate is showing. Therefore, the value of nextstate should be one of the active states and its domain is the same as the domain of state excluding ready and wait.
- **pevent** is a variable for keeping the list of events for which the process is waiting at each time it is in wait state. In PicOS applications, before a process releases the CPU and goes to wait state, it can set some events as "interesting" ones. If any of these events are triggered, the process should be invoked and its state should be changed to ready, i.e. whenever the process gets the CPU it can run in the active state shown by nextstate. This set of events is recorded in pevent. In our model we defined pevent as a ten-bit word, which means the maximum number of possible events that a process can wait for is ten. The size of the pevent word should be the same as event word, since for each event/interrupt defined in the program we need a bit in pevent. Therefore, the maximum number of the events a process can wait for is the number of the defined events/interrupts.
- **pstate** is a variable which shows the state in which a process should be woken up if an interesting event (which is identified in pevent) is triggered. In PicOS applications, when a process wants to wait for some events, it should set the state in which it is interested to be woken up if each of those events is triggered, e.g. wait(e1, s1) means if event e1 occurs, the next active state in which the process should be woken up is state s1. This set of states is recorded in pstate. Since for every event for which the process may wait, an active state should be set, the size of pstate should be the same as the size of pevent. In addition, the value of each member of pstate can be one of the active states; therefore, pstate should be defined as an array of active states, e.g. array 1..5 of $\{$s1, s2, s3, s4$\}$.
- **timer** is a variable which shows the amount of time which is left till a time event be triggered. In PicOS applications, other than events that are set in pevents, processes can wait for a time event, e.g. delay(t1, s1) means after t1 milliseconds if the process is still in wait state, a time event will be triggered for the process and it become ready to go to state s1. This means that the process sets a time value, and when it goes to the wait state, the timer counts backward until it reaches the value 0, meaning that it is time for the time event to be triggered. Note that each process can wait for no more than one timer event at a time and the value of the timer variable shows how many clock ticks are left until that time event triggers.
- **tstate** is a process variable which shows the state in which a process should be woken up if the timer event is triggered. Same as pstates for pevent, tstate should be set for time event, containing the state in which the process wants to be woken up after recognizing the time

event. For the same reason we explained for pstate, tstate should be one of the active states of the process, but since we only can wait for one timer event at a time, tstate is a variable of type active events (not an array of them).

- **lock** is a process variable which shows if a process is running or not. The value of lock is true if and only if the process is in one of the active states (any state except wait and ready). This variable is used by other parts of our model to ensure that at any time only one of the processes of the node owns the CPU.

**Implementation Guidelines**

1. **Initializing process variables**

   In the first step of building a process, its variables should be initialized as a new FSM in PicOS is being created. Recalling from the modeling assumptions, each FSM in the PicOS application is in the ready state when the application starts running. Therefore, we need to initialize the variables so that our model seems like a ready process that is not waiting for any kind of events and can start running from the start state whenever it gets the CPU. The initial value of each of the variables and parameters of process will be assigned as follows:

   - Variables timer, state, pevent, nextstate, pstate and tstate should be defined inside the VAR block of NuSMV model. Their initial value should demonstrate a new ready process:
     - init(timer) :=0; which means the process is not waiting for any time event.
     - init(state) := ready; which means as soon as it is the process' turn to run it can start running.
     - init(pevent) := $0b10\_0$; which means the process is not waiting for any timer event.
     - init(nextstate) := s1; which means as soon as the process gets the CPU it wakes up at state s1; where s1 is the start state of the process, i.e. the first state defined in the process.
     - init(tstate) := s1, and init(pstate[i]) := s1; where s1 is the start state, and i can be integers from 1 to the size of pstate. Hence that tstate and pstate will be ignored at the beginning since timer and pevent variables show that the process is not waiting for any event; process is not waiting for anything, so the value of the tstate and pstate is not actually matter. However, since we want to have a single start point for the model checker we initialize them to the start state. Recall from NuSMV constraints, we cannot use pstate[i] where i is a variable itself; therefore, for initializing pstate we should actually initialize each element of it individually, e.g. init(pstate[1]) := s1; init(pstate[2]) := s1; etc.
   - Variable lock needs to be updated based on the state of the process; therefore, lock can

initialize itself based on the value of the other initialized variables at first step. We decided to define lock in the DEFINE block of NuSMV model, since there is only one rule that is used for evaluating its value, and that is based on the value of state, i.e., lock is true if state is one of the active states; lock := state=s1 | state=s2 | state=s3 | state=s4;

- Parameters pid, currentId, event are the input parameters that are begin passed to process element, they should be initialized in the element that instantiates process, i.e. PCB. Therefore, initializing those parameters is not the process' concern.

2. **Non-Time Event Handling**

Sometimes we want the processes to wake up in a specific state whenever a specific event is triggered. In PicOS programming, this can be done by calling a when(EVENT, STATE) command in an entry of the FSM before it releases the CPU. This command means that this process is interested in EVENT, and after the process goes to wait state, if EVENT is triggered, the process will wake up in state STATE. Through this command each process can set several events as interesting ones, and for each of these events the state in which it wants to be woken up should be set. To model the *when* command, we used pevent and pstate variables which are respectively responsible for recording the interesting events of a process and the states in which it wants to be woken up if any of those events is being triggered.

```
next(pevent) :=
  case
    state = wait&((event & pevent) != 0b10_0 | timer=1) = 0
        b10_0;
    state = s1 : 0b10_111;
    state = s2 : 0b10_10;
    TRUE : pevent;
  esac;

next(pstate[1]) :=
  case
    state = s1 : s2;
    TRUE : pstate[1];
  esac;

next(pstate[2]) :=
  case
    state = s1 : s1;
    state = s2 : s4;
  TRUE : pstate[2];
  esac;

next(pstate[3]) :=
  case
    state = s1 : s3;
    TRUE : pstate[3];
  esac;
```

Listing A.1: Non-Time Event Handling Example

Listing A.1 shows how we modeled when(EVENT, STATE) statements of a PicOS application using pevent and pstate. Followings are the list of properties that are considered in this model:

- Each process can be waiting for several events in each state. For example, the line state = s1 : 0b10_111 shows that if the process is in state s1, pevent should be set as 111, which means it wants to wait for the first, second and third events. It also means that we should set the *wake up* states for these three events. Since pstate is an array, for setting relative states for the first, second and third event, we should modify pstate[1], pstate[2] and pstate[3].
  According to this explanation, the commands that we modeled in these steps are when(e1,s2), when(e2,s1), when(e3,s3) that have been written in code of s1 entry of this process, and e1, e2, and e3 are the first, second and third events.
- A process can set a specific event as an interesting one for more than one entry. For instance, in a single process we can have when(e1,STATE) in both s1 and s2 entries. The same thing is modeled in Listing A.1 where we have the command when(e2,s1) in s1 entry, and we have the command when(e2,s4) in s2 entry.
- Whenever any of the events that the process is waiting for is triggered, we update the value of nextstate by using the related state, and then empty the pevent. The first part is done by logics for setting nextstate, but the second part (flushing pevent) is modeled by resetting pevent. Since the events for which the process is waiting could be from the list of pevents or a timer event, we need to flush pevent if either of these events is triggered. In other words, pevent should be flushed if the intersection of the list of triggered events (event variable) with the list of interesting events (pevent variable) is not empty, or if timer event is about to be triggered (timer = 1). Listing A.1 shows how we modeled this behavior using state = wait&((event & pevent) != 0b10_0 | timer=1) = 0b10_0;
- Based on which interesting event is triggered, the value of pstate affects the nextstate of the process. This feature is handled while modifying nextstate. Here we based our model on the point that if the first event occurs, the value of nextstate should be the same as pstate[1], if the second event occurs, the value of nextstate should be the same as pstate[2] and so on.

Based on these points, for modeling when (E, S) in an entry (e.g. STATE) of a process in PicOS application, we should:

- Add a line to the logic of pevent changing, e.g. state=STATE : 0b10_ 1 if E is the first event, or state=STATE : 0b10_10 if E is the second event and so on (if we have several when statements, we can have something like state state=STATE : 0b10_11 if the interesting event are the first and second one).

- Add a line to the logic of the corresponding pstate, e.g. add state=STATE : S to pstate[1] if E is the first event, or add state=: S to pstate[2] if i is the second event and so on.
- Keep the generic parts such as the rules for flushing the pevent the same.

3. **Time Event Handling**

Sometimes we want the processes to wake up in a specific state whenever a timer event is triggered. It is the same as what we explained for pevent, but here we are waiting for a time event. In PicOS, time event can be set using the command delay(CLOCKS, STATE), which means we want our process to stay idle for CLOCKS steps after releasing the CPU and then wake up at STATE; moreover, the process should not wake up sooner except if another interesting events is triggered. Since the way a timer event is handled in PicOS is slightly different from the way the other events are handled, we also modeled time event differently. Here we employed timer and tstate variables.

Listing A.2 shows how we modeled delay (CLOCK, STATE) statements of a PicOS application using timer and tstate variables. Followings are the list of properties that are considered in this model:

```
next(timer) :=
  case
    state = wait & ( (event & pevent) != 0b10_0 | timer=1) :
        0;
    state = s1 : 100;
    state = s2 : 50;
    timer != 0 : timer - 1;
    TRUE : timer;
  esac;
next(tstate) :=
  case
    state = s1 : s3;
    state = s2 : s4;
    TRUE : tstate;
  esac;
```

Listing A.2: Time Event Handling Example

- Each process can wait for no more than one time event in each state. It means that if there are several delay statements in a single entry of a process, just one of them is considered. In this situation, we should consider the last delay statement. This means we just aim to model the last delay statement of each entry. As an example of modeling the delay command, state=s1:100 shows that if no other interesting event is triggered, the process will become ready after 100 clocks. It also means that we should set the state we want to be woken up after the timer event. Therefore, state = s1 : s3 shows that when the process releases the CPU after being in state s1, it wants to wait for a time event and when that event is triggered it want to be woken up at state s3. According to this explanation, the command that we modeled in these steps is delay(100,s3), that has been written in s1 entry of this process.

- A process can be programmed to wait for time event in more than one entry and the amount of time that the process should wait for time event can be set to a different value in each of those entries. The same thing is modeled in Listing A.2 where we have the command delay(100,s3) in s1 entry, and have the command delay(50,s4) in s2 entry.
- Each time an event that the process is waiting for comes, the process will become ready. Therefore, the process should no more wait for any time event to be woken up. So, same as flushing pevent, we should reset the value of timer and change its value to 0 (state = ready : 0).
- In each step, if the value of timer is not zero, we should decrease its value by 1. This means that at each step a clock passes so the process should remain waiting for the time event one clock less (timer != 0 : timer - 1).

Based on these points, for modeling delay(C, S) statement in an entry (e.g. STATE) of a process in PicOS:

- Add a line to the logic of timer changing, e.g. add state=STATE : C where C is an integer stating the number of clocks which the process should wait till receiving the timer event.
- Add a line to the logic of tstate, e.g. add state=STATE : S.
- Keep the generic parts such as the rules for flushing the timer or decreasing its value the same.

Note that our model may not exactly behave the same as PicOS application in this case, and the main reason is that here we assumed our time to be discrete which is not the same as the real world. Besides, the amount of time it takes our model to perform the state transitions and triggering the events is always one clock; whereas, the same activities may need several clocks in a real PicOS application. Therefore, the sequence of events which are triggered in our model may be different from what happens in the real PicOS application.

4. **Deciding Next Running State**

Nextstate of a process shows the next running state in which the process should be woken up. The value of this variable can be affected in two different ways: the states which are set in pstate or tstate, and the PicOS command proceed. In the following we explain how these factors can affect nextstate of the process and how we model the expected behavior.

- A process can wait for time event or any other event which is set in its pevent variable. If any of these interesting events is triggered, the process should be woken up in a defined running state which is set as their tstate or pstates. Therefore, we should always check to see if the incoming events are one of those interesting events. If the intersection of triggered events and the interesting ones contains the first event, we should set the wake-up state (nextstate) as pstate[1], else if it the intersection is the second event, the wake-up state should be set as pstate[2], and so on. Besides, if the value of the timer is equal to 1, it means that in the next step the timer event will be triggered, so the wake up state should be tstate. This checking is modeled as it is shown in Listing A.3 (state =wait & (event & pevent & 0b10_1) != 0b10_0 : pstate[1];). Note that the order in which we check these conditions should be the same as the priority of the events. Here we assumed that the priority of first event is more than the second one, and so on, and all of them have higher priorities than the time event.

- The command proceed(STATE) can affect the state in which the process should be woken up next time it gets the CPU. This means that the process should release the CPU and its state should be set as ready without waiting for any kind of events. Then whenever the process can take the CPU, it wakes up at STATE. Therefore, proceed command should set the next running state as STATE. For example in Listing A.3, state = s3 : s2 shows that at the end of s3 entry of a process in PicOS we have proceed(s2) command. Note that this is just part of modeling of proceed. The remaining details in this regard is included in state transitions procedures.

```
next(nextstate) :=
  case
    state=wait & (event & pevent & 0b10_1)!=0b10_0: pstate[1];
    state=wait& (event & pevent & 0b10_10)!=0b10_0: pstate[2];
    state=wait&(event & pevent & 0b10_100)!=0b10_0: pstate[3];
    state=wait & timer=1 : tstate;
    state=s3 : s2;
    TRUE : nextstate;
  esac;
```

Listing A.3: Deciding Next Running State Example

5. **State Transition**

Transitions between the states of a process result in the behavior of a node. Based on the entries that are defined in the process, the sequence of the triggered events, and the commands used in coding those entries (such as when, delay, proceed, sameas), the transitions between states can vary. In this regard, we considered the following and modeled the state transitions based on them as it is shown in Listing A.4.

- When the process is in wait state, and it is waiting for some events (time event or other types of events) which are triggered, then the state of the process should be set to ready (state = wait & ( (event & pevent) != 0b10_0 | timer=1) : ready). The reason is that the process has been waiting for an event and now the event has occurred, so the process should not remain in wait state anymore. Now it should be ready till when it gets the CPU and it can be executed.

- When the process is ready, and it is its turn to be run, its state should be set as nextstate which has been set as the desired state for process for being woken up in (state=ready & pid=id : nextstate;). Here pid=id is the case that shows it is this process' turn to run. More detail about this behavior is explained in the following section.
- Whenever the command release has been called at the end of a PicOS entry, we should change the state of the process from that entry to wait. Therefore, release means change the value to wait (state=s1 | state=s2 : wait;).
- Whenever the command proceed(STATE) has been called at the end of a PicOS entry, we should change the state of the process from that entry to ready (state=s3 : ready). This is because proceed means to release the CPU, but stay ready till the process' turn comes and it can takes the CPU and run.
- Whenever the command sameas(STATE) has been called at the end of a PicOS entry, we should change the state of the process from that entry to STATE (state=s4 : s3). The reason is that sameas command means to jump to the STATE that has been set.

```
next(state) :=
  case
    state=wait & ((event & pevent)!=0b10_0 | timer=1) : ready;
    state=ready & pid=id : nextstate;
    state=s1 : wait;
    state=s2 : wait;
    state=s3 : ready;
    state=s4 : s3;
    TRUE : state;
  esac;
```

Listing A.4: State Transition Example

6. **Running one process at a time**

In PicOS applications, only one of the processes of a node can take the CPU at any time. This means that no more than one process of a node can be activated simultaneously. In our model, we control this behavior by using id and pid in each process.

As we explained, each process has a pid, and all of the processes can see the id of the current process. So, the only thing that should be performed in each process is to compare pid and id. This is how this comparison can help us achieve this goal:

- In each process, if pid = id, the process is the same as the current process and it can be woken up and change its state to the desired running state if it is ready at that time. This also means that the process changes its state to one of the running states and the lock value of the process is changed to true. Note that if the process is not ready, nothing happens even though we have pid=id.

- In each process, if pid != id, the process and the currently running process are not the same. Therefore, the process is not allowed to wake up even though it may be ready. In this situation, the process should stay idle till id changes and pid becomes equal to id.

Listing A.4 shows where we use this comparison in one of the conditions of changing the state of the process. As pid is a unique numbers, and id changes in range of existing pids, at any time we have pid=id for just one of the processed, and we have pid!=id for all other ones. Therefore, only one process that is ready can take the CPU and start running at any time. Accordingly, we can say that at any time at most one process is running, and it is the behavior we wanted to model.

## A.2 PCB (Process Control Block)

### Detailed Design

The PCB is modeled as a singleton module in NuSMV and has the following variables:

- **Process instances** are instantiated and kept in PCB. Each process, as a variable for PCB, needs a name to be known with inside PCB. While instantiating each process, PCB should pass a unique number to it as its process identifier. The PCB just initiates the processes, and will not change any of their values at all.
- Every parameter that is needed for instantiating a process is passed to it by the PCB; therefore, PCB should know about those variables too. For example, **event** and **currentId** parameters are required for computations in some processes; however, these parameters are defined outside of each process and are needed to be passed to them somehow. To achieve that goal, PCB acts as a facade for the processes, it receives those parameters while being initiated, and pass them to its including processes while defining them. As a result, except for process id which PCB selects for each process, all the other parameters that are required for instantiating a process should be passed to PCB while defining it.
- **lock** is a PCB variable which indicates if any of its processes are running or not. The value of lock is true if and only if one of its process is in one of its active states and it is false if all of the including processes are in ready or wait state. This variable is used by the scheduler element so that it can decide what strategy to use for selecting the next process to run. For example, if PCB lock is going to be true it means that one of the processes are running in the next step and the scheduler should not select any other process to run.

**Implementation Guidelines**

1. **Instantiating Processes**

   As it was explained before, the PCB is basically responsible for instantiating all processes that are defined in the application. Each process can be instantiated by receiving three parameters, the assigned identifier (pid), the current running process identifier (currentId), and the set of triggered events (event). Among these parameters, the first one is considered as a unique number for the process and the others are the same for all processes of the node. We modeled PCB as a module that receives two parameters, id and event. Consequently, having these two pieces of information, we can instantiate as many processes as needed for the node just by assigning unique process identifier (pid) to each process.

   It is also notable that although PCB is a list of processes, it cannot be modeled as an array of processes. The reason is that each process is modeled as a module in NuSMV, and therefore an array of processes can be seen as an array of modules, which according to the NuSMV limitations the type array of module is not supported by NuSMV.

   An example of a PCB model is shown in Listing A.5. In this example, PCB contains four process instances, three instances of a process1 and on instance of process2. Note that we can have as many instances of a process that is needed, as long as we give them unique identifiers. Besides, all instantiated processes can be from the same type or we can have instances of different types of defined processes.

```
MODULE pcb (id, event)
  DEFINE
    lock := p0.lock | p1.lock | p2.lock | p3.lock;
  VAR
    p0 : process1(0, id, event);
    p1 : process1(1, id, event);
    p2 : process1(2, id, event);
    p3 : process2(3, id, event);
```

Listing A.5: Process Instantiation Example

2. **Defining PCB Lock**

   One of PCB responsibilities is to provide information to the outside elements, specifically scheduler, and inform them if any of the processes are running or not. It is important to keep track of the states of the involving processes in PCB. However, from the scheduler's viewpoint, we just care if any of the processes is running or all of them are idle (not running). In other words, we just want to know if a process is *locked* (its lock is true) or not. If any of the processes is running (is locked), we do not care which one of them is running, but the only thing that matters is that the CPU is assigned to one of the processes, and not to the scheduler. Accordingly, the lock variable should be defined in the PCB to facilitate accessing to the information about the processes running states. Its value shows if any of the processes is running or all of them are idle. In other words, the value of lock is true if and only if one of the processes is running, and its value is false if and only if all of the processes are in ready or wait states (but not in any of the active states). Therefore, the value of PCB lock is based on the value of involving processes' locks, and it should be updated at any time of the execution. All these considered, the PCB lock should be defined in the DEFINE block of the NuSMV model using the following rule:

$$lock = \bigvee_{p \in PCB} p.lock$$

In the formula p stands for a process, p.lock means the lock variable of process p. The rule means lock is the result of the logical OR operator between the lock variables of all processes in the pcb. Therefore, lock is false if and only if all of processes' locks are false (all processes are idle), and it is true if one of processes locks is true (process is running).

In order to implement the rule in NuSMV, we need to perform the OR operator on all of the process locks, which means that if PCB contains four process, say p1, p2, p3, p4, the PCB lock is defined as follows:

$$lock \ := \ p1.lock \mid p2.lock \mid p3.lock \mid p4.lock;$$

Note that although this lock could be defined in the scheduler or any other module that has the access to the processes, we decided to define the it in PCB since it was the closest to the processes, and it was not needed to make process locks visible to any other module.

## A.3 Scheduler

**Detailed Design**

The data needed by scheduler depends on the strategy it is implementing. Since our selected strategy is a regular iteration on processes (it resets every time a new process is begin scheduled) and setting the current running process id, we only need the information about the processes and the variable that shows the current running process id. Like the other elements in our model, scheduler is also a NuSMV module, with one instance of it for each node of the model. In our schema, the selected design for scheduler module contains the following data:

- **currentId** is the identifier of the process that is currently running, a.k.a. current process id. This variable is needed to be set by the scheduler, however, since it is one of the key variables in the model and other elements repeatedly need to have the access to read it., we decided to define the variable outside of scheduler, and pass the variable to scheduler as a parameter. Therefore, scheduler gets the raw variable of currentId and is responsible for initializing it and setting its value at each step.

- **pcb** is another parameter which needs to be passed to scheduler while defining it. pcb is an instance of PCB element and contains the list of all processes and can also be considered as an array of processes over which the scheduler iterates. Scheduler needs to know about the value of lock variable in pcb and select the next CurrentId based on that. e.g., if PCB lock is going to be true it means that one of the processes are running in the next step and the scheduler should not select any new one to run.

### Implementation Guidelines

According to PicOS documents, scheduler should walk through PCBT to see which one of its processes is ready to be run. As the scheduler finds a ready process, it selects it as the next running process, which means CPU will be given to that process. Whenever scheduler gets the CPU again, it starts from the beginning of PCBT and searches for the first ready process.

In order to model this behavior of the scheduler, we employed pcb as the array of processes, CurrentId as the index of the current process in pcb, and the lock of pcb which shows if any process is running or not. In this part of our model, we just explain how to iterate over the processes of the pcb, but checking if a process is ready or not is done in process module.

Iterating over processes of pcb is the same as changing the value of CurrentId, indicating the process which will be checked to see if it is ready or not. Following is the list of what we expect from our scheduler in iterating over pcb and changing the CurrentId, and how we modeled it in NuSMV. The complete model covering all these expectations is shown in Listing A.6.

```
MODULE scheduler(currentId , pcb)
  ASSIGN
    init(currentId) := 0;
    next(currentId) :=
      case
        next(pcb.lock) : currentId;
        pcb.lock : 0;
        TRUE : (currentId+1) mod 4;
      esac;
```

Listing A.6: Scheduler model Example

- CurrentId should not be changed when any of the processes is running. The reason is that, when a process is running, it means the process owns the CPU at that time. So, the scheduler does not own the CPU and it is not capable of changing the CurrentId at that moment.
  We modeled this feature using the pcb lock which shows if any of the processes are running or not. If the next value of the lock is true, it means that during next clock the CPU is still owned by one of the processes, so scheduler cannot change the CurrentId (next(p.lock) : CurrentId;).
- Whenever Scheduler takes the CPU after any of the processes finished running, it should start iterating over pcb from the beginning of the list. This means that if none of the processes are

running in the next step, and if scheduler owns the CPU in the next step, we need to choose a new value for CurrentId and that value should be 0.

We modeled this feature by using the current and next values of pcb lock. The actual condition that results in resetting the CurrentId is that a process is running in current step (p.lock = true) but it will finish running and release the CPU in the next step (!next(p.lock) = true). Therefore, the combination of these two conditions should result in resetting the CurrentId and starting iteration from the beginning of the list (!next(p.lock) & p.lock : 0;).

- If CurrentId is not pointing to a ready process, scheduler should continue iteration by increasing the value of CurrentId. As we explained before, whenever the current CurrentId is equal to pid of a process which is ready to be run, the process gets the CPU and runs, and if the process is not ready it does nothing. Therefore, we should change CurrentId to give the chance of accessing CPU to other processes. This means that if none of the processes is running (and none will run till next step), the process that is pointed by CurrentId has not been ready yet. Therefore, we should go to the next CurrentId in the list (!next(p.lock) & !p.lock : (CurrentId +1) mod numberOfProcesses;).

## A.4 SharedVariables

**Detailed Design**

We defined the sharedVariable module that is responsible for organizing modifications of the variables that are shared between processes, e.g., event. The properties that should be accessible to SharedVariable module to make the required functionality possible to be done plus the required variables or parameters in SharedVariable module are listed as follows:

- **event** is the shared variable between processes that should be managed and assigned, therefore, at the first place the module needs to have the access to that. However, since event is repeatedly needed by other elements of the model, we decided to define it outside the SharedVariables and just pass it to the SharedVariables module for being assigned.
- **pcb** which contains a list of all processes is needed as a passed parameter to SharedVariables. We need the reading access to processes in order to to read the values of their local variables and decide about the value that should be assigned to the shared variable, i.e. event.
- **interrupt** builds part of the event variable, defined in node and network levels, but it needs to be set in a lower level. Triggering interrupts is an inter-node communication concern, as event triggering is a intra node concern. Therefore, interrupts can be passed to SharedVariables module just to be used in connecting it to the internal events and building the whole event variable.

In our model, an event is any kind of incoming signal which a process or node may be interested in. Events can be triggered by processes of a node, or it can be received as interrupts that comes from an external nodes or other devices in the environment. Note that although event results from the concatenation of interrupts and internal events, we assume that the shared module which is defined for a process is just responsible for modifying the internal events. Therefore, interrupts are handled with the same approach but from outside of a node. So, at this level, interrupts just needed to be passed to the sharedVariable module as input parameters.

Another point which worth noting is that, in our modeling schema, we assumed that the only shared variable is event variable, however our model can be extended in this regard and we can add more shared variables. The only necessary step for this extension is to pass the shared variables as parameters of sharedVariables module when instantiating it, and then define the related conditions for deciding about the values that should be assigned to them.

### Implementation Guidelines

1. **Internal events**

   Within a single node, processes may wait for a specific event that can be triggered by another process of that node. We refer to those events as internal events, since they affect the internal behavior of the node, and cannot be seen by other nodes in the network. Therefore, the context of internal events is a single node.

   The number of internal events can vary from node to node, because it depends on the number of possible events that are defined in a node and can be triggered by the internal processes. In our models, as it was specified in the modeling assumptions, we assumed that the number of possible internal events is five. So, we defined internal as a five-bit word, which means we allocated five bits for internal events.

   The way we decide about how to change the internal events, i.e. organizing their changes, directly comes from the idea of defining sharedVariables module for shared variable modifications. Here we have internals as the shared variables, since all of the processes in the node can trigger internal events. Since each process may trigger a different type of event depending on the state in which the process is running, we should set the value of internals, based on these different conditions.

   Listing A.7 shows examples of the conditions that should be considered in assigning a value to internal events.

```
MODULE shared(pcb , event, interrupt)
  VAR
    internal : word[2];
  ASSIGN
    init(internal) := 0b5_0;
    next(internal) :=
      case
        next(pcb.p0.state) = s1 : 0b5_1;
        next(pcb.p1.state) = s2 : 0b5_11;
        TRUE : 0b5_0;
      esac;
    event := interrupt::internal;
```

Listing A.7: SharedVariables Model Example

For instance, next(pcb.p0.state) = s1 : 0b5_1 shows that process p0 triggers the first event when the process is in state s1, or next(pcb.p1.state) = s2 : 0b5_11 shows that process p1 triggers both the first and the second events when the process is in state s2. Besides, TRUE : 0b5_00 shows that if none of the conditions is held it means that no event is triggered, so the value of internal should be 0. Note that, these conditions can also be based on the value of some variables other than states. For example, assume that there is a variable named x which is defined in a p0, and we want the first event to be triggered only if p0 is in state s1 and the value of x is equal to "blah", therefore we can model this as a case that says next(pcb.p0.state)=s1 & next(pcb.p0.x) = blah : 0b5_1. Accordingly, the conditions deciding the value of internal events depend on the internal logic of the processes of the PicOS application. The guideline for extracting these conditions out of the PicOS application is as follows:

*Assume p is a process and s is one of its states, if somewhere in the s coding block event e is triggered, the sharedVariable element of the model will have next(pcb.p.state) = s : 0b5_01; where e is assumed as the first bit of the event variable.*

2. **Interrupts**

   Sometimes the processes are waiting for events which may be triggered by processes of other nodes. We refer to these kinds of events as interrupts which are mainly based on the multi-node context of PicOS applications.

   Interrupts are shared between nodes of a network. So modifying it can be done outside of the nodes (see Section A.6). Therefore, here we have interrupts as an input parameter for shared module. Accordingly, the only thing that should be done is concatenate the value of interrupt and internal events to build the main event.

## A.5 Node

**Detailed Design**

According to the node responsibilities explained in the above, node element can be a combination of smaller elements and parameters. It can be designed as a NuSMV module that receives the shared variables between nodes (i.e. interrupts) as a parameter and pass it to its elements which want to use it (shared). Since scheduler of a node is responsible for controlling the activities of the node, it needs to access to other elements of the node (such as pcb, shared). Therefore these variables should be defined in the node module. Following is the list all data needed in node element:

- **CurrentId**, which is the identifier of the process that is currently running, or the current process id. The domain of this variable shows the maximum number of processes in the node, but its value shows the index of the running process in pcb. The variable defined in the node to be passed to pcb and its processes, but scheduler is the element responsible for setting its value.
- **event**, which involves the shared variables between nodes or processes. Event is the result of the concatenation of interrupts and internal events. Note that here we modeled event as a ten-bit word (five bits for interrupts and five bits for internal events), however this size can be modified based on the PicOS application we are modeling. The only important thing in this regard is that size of event should be equal to the size of interrupts plus the size of internal events.
- **pcb**, which contains a list of all processes and can be considered as an array of processes over which the scheduler iterates. This element needs currentId and event as parameters in order to pass them to its involving processes while instantiating them.
- **sharedVariables**, which is the shared variable controller and organizes internal event modifications. This element needs pcb to access the internal variables of its processes (such as their states), event to modify the bits which are related to internal events, and interrupt to concatenate it with internal events and result the main events.
- **interrupts**, which is a parameter passed to the node element while being instantiated by the network element. This parameter builds part of the event variable and is needed to be passed to sharedVariable element which is responsible for organizing events. Since interrupts are shared between nodes of the network they should be defined and modified in network layer.

**Implementation Guidelines**

Node element is responsible for building the structure of the actual node, containing an instance of PCB, sharedVariables and scheduler. In addition, node should be capable of receiving interrupts. The former is possible by defining the variables and passing the required parameters to them; whereas the latter can be handled by getting the interrupt as an input parameter. Accordingly, node is just a structure of its inner elements which can be initiated as shown in Listing A.8.

```
VAR
  id : 0..3;
  event: word[5];
  p : pcb(id, event);
  s : sharedVariables(p , event);
  sch: scheduler(id, p)
```

Listing A.8: Node Model Example

- currentId should be defined as a variable of node, so that it can be passed to the scheduler for being initialized and modified, and to PCB for being used by its processes. The domain of the currentId shows the number of processes in the node, i.e. the size of pcb.

- events should be defined as a NuSMV word, a set of bits, of which size is the sum of the internal events and interrupts defined in the application. Here we assume that 5 bits are defined for internal events and 5 bits for interrupts, therefore, the events should be word of size 10.
- pcb is an instance of PCB element which needs currentId and events as the input parameters. There should be only one instance of pcb in a node.
- sharedVariable is an instance of sharedVariables element which needs pcb, event and interrupts as the input parameters. There should be only one instance of sharedVariable in a node.
- scheduler is an instance of scheduler element which needs pcb and currentId as the input parameters. There should be only one instance of scheduler in a node.

## A.6 Network

**Detailed Design**

Considering the target responsibilities for the network element, the network module needs to include the following variables:

1. **Node Instances** are initiated and kept in network. Each node, as a variable of network, needs a name to be known with, and the variables shared between nodes, i.e. interrupts, should be passed to all of the them while instantiating them.
2. **Interrupts**, which is a parameter passed to the node element while being instantiated by the network element. Since interrupts are shared between nodes of the network they should be defined and modified in network layer. Interrupts is a word of size n, where n is the maximum number of message types that can be transferred between nodes (one bit for each message type).

**Implementation Guidelines**

1. **Instantiating Nodes**

   A network cannot be built without defining its nodes and the connections (communication channels) between them. Therefore, the first responsibility of network it to define interrupts as communication channels and instantiate nodes by passing the interrupt as their input parameters. Note that we pass interrupts to nodes since they use it as part of their events in their shared modules.

   Listing A.9 shows a simple network containing two nodes which are connected using an interrupt. Here we defined interrupts as a five-bit word, means we can handle a set of five different interrupts in this model. However, the size of interrupt can be changed if needed. The only important thing in this regard is that changing this size may affect the size of the event which is defined in each of the nodes; since the size of event should be equal to the size of the node's incoming interrupt plus the size of its internal events.

```
MODULE main
  VAR
    interrupt : word[5];
    NODE1 : node1(interrupt);
    NODE2 : node2(interrupt);
  ASSIGN
    init(interrupt) := 0b5_0;
    next(interrupt) :=
    case
      next(NODE1.pcb.p0.state) = OFF : 0b5_10;
      next(NODE2.pcb.p1.state) = OFF : 0b5_1;
      TRUE : 0b5_0;
    esac;
```

Listing A.9: Network Model Example

2. **Organizing Interrupt Modification**

   As it is shown in Listing A.9 , interrupt is a variable which is shared between two nodes and may change based on the value of each node internal variables. Therefore, assigning values to interrupts cannot be done in any of the nodes, but it should be done in another module which has the access to all the nodes, i.e. the network. Accordingly, one of the responsibilities of the network module is to organize interrupt modifications and decide about the new values that should be assigned to the interrupt.

   The way we handle changing the interrupt is the same as what we explained for organizing changes for events. The only difference is that here the shared variable is interrupt, and the components which share the variable are the nodes of the network. Therefore, similar to what we had in shared module, since each node may trigger a different type of interrupt depending on in the state of its processes or its other variables, we should set the value of interrupt based on these different possible situations.

   Listing A.9 shows examples of the conditions which should be considered in assigning a value to the interrupt variable. For instance, next(NODE1.pcb.p0.state) = s1 : 0b5_10 shows that if the p0 process of NODE1 is running in state s1 the value of interrupts should be 0b5_10, which means the second type of interrupt would be triggered for all of the nodes that are connected to this channel, or next(NODE2.pcb.p1.state) = s2 : 0b5_1 shows that if the p1 process of NODE2 is running in state s2 the value of interrupts should be 0b5_1, which means the first type of interrupt would be triggered for all of the nodes that are connected to this channel. Besides, TRUE : 0b5_0; shows that if none of the above mentioned conditions is held, the value of interrupt should be 0, which means no interrupt has happened.

# Appendix B

# Translating ArduinOS Programs Into NuSMV Models

Much of our experience in the previous PicOS modeling exercise can be reused. For each application, the corresponding NuSMV model must contain node, pcb, scheduler, shareVariables and all processes defined in the program. The network element, however, has no equivalent in the node program, since its logic resides in the connections between nodes. In other words, the information about the network of the nodes is actually provided by wiring up the Arduino nodes together. The hardware connections need to be specified and be modeled separately.

We now outline the general process of building the NuSMV model out of the source program of an ArduinOS representation (.node file).

1. A NuSMV module is defined for each of the elements in the kernel layer: node, PCB, Scheduler, and SharedVariable. A module is also created for each process in the program. The required parameters for each module will be described in the next steps.
2. Module node definition
   (a) For each i/o pin defined in the program, a variable should be passed to node module as an input parameter, and the node passes it to all of its inner elements while instantiating them. If there is no i/o pin defined in the program, then the node module has no input parameter, and also does not have any way to communicate with other nodes in the environment.
   (b) Set the signature of the node element by generating a name and a set of parameters. The name of the node element will be "node_nodeName" where nodeName is the name of the node in the source program. Secondly, its input parameters should be set as all the input or output pins that are defined in the source file.

   MODULE node_nodeName(inputChannelName*, outputChannelName*)

   (c) Define the variables of the node module in its VAR block as follows:
   - currentId is an integer showing the current running process identifier. Therefore,

its domain is from 0 to N-1 where N is the number of processes instantiated in the "root" method of the source file, e.g. currentId : 0..3.

- event is a word of size N bits, where N is the number of events defined in the node of the source file, e.g. event : word[2];. Note that the events are numbered as their order of being defined. For example, if two events are defined in the application, the first bit of the event word stands for event e1 (0b2_01), and the second bit stands for event e2 (0b2_10).
- For each variable defined in the node scope of the program, a variable is defined in VAR block of node element and is passed to node inner elements, i.e. pcb, sharedVariable, scheduler [1]. Note that the *output* variables will be set by sharedVariable module, since they are shared between all processes of a node.
- PCB should be instantiated using the signature of the PCB module and should pass the needed parameters to it. We called the PCB instance as "pcb" and passed currentId, event, all the variables defined in the node, and all defined input pins to the node_nodeName_PCB module, which is the PCB element of the model for node_nodeName.
  pcb : node_nodeName_PCB (currentId, event, nodeVariable*, inputChannelName*);
- Scheduler should be instantiated using the signature of the Scheduler module and the needed parameters should be passed to it. We called the Scheduler instance as "scheduler" and passed currentId and pcb as the parameters to the node_nodeName_Scheduler module, which is the scheduler element of the model for node_nodeName.
  scheduler : node_nodeName_Scheduler(currentId, pcb);
- SharedVariables should be instantiated using the signature of the sharedVariables module and the needed parameters should be passed to it. We called the SharedVariables instance as "sharedVariables" and pcb, event, all the variables defined in the node, and all defined input and output pins as the parameters to the node_nodeName_SharedVariables module, which is the sharedVariables element of the model for node_nodeName. We are passing variables of the source node program and the defined outputs to sharedVariable since this element is actually responsible for setting values to the outputs.
  sharedVariables : node_nodeName_SharedVariables(pcb, event, nodeVariable*, outputChannelName*, inputChannelName*);

3. Scheduler module definition

   (a) For each variable defined in the node scope of the program, a variable should be passed to PCB module as input parameter, and pcb should also pass it to to all of its inner processes, while instantiating them.

[1]pcb element should also pass these parameters to to all of its inner elements,processes, while instantiating them.

(b) Set the signature of the Scheduler element. The name of the module will be "node_nodeName_Scheduler" where nodeName is the name of the node in the source program. The parameters that should be passed to the module are currentId and pcb.[2]

MODULE node_nodeName_Scheduler(currentId, pcb)

Listing B.1 shows the completed model of the scheduler element for a node containing 3 processes. Note that in Arduino the scheduler iterate over processes in order and schedules them one by one, whereas in PicOS every time a new process should be scheduled the scheduler starts iterating over pcb from its beginning.

```
MODULE node_N0_Scheduler(currentId , pcb)
ASSIGN
  init(currentId) := 0;
  next(currentId) :=
  case
    next(pcb.lock) : currentId;
    TRUE : (currentId + 1) mod 3;
  esac;
```

Listing B.1: ArduinOS Scheduler model Example

(c) Assign a value to currentId in order to simulate iteration over processes included in pcb. In the ASSIGN block of the scheduler, we should initialize the value of currentId, and decide about its next value at each step. Since currentId is trying to simulate a pointer to a process in the processes list of pcb, for initializing it we should set it as pointing to the first process in the list, init(currentId) := 0;.

The next value of currentId should be set somehow that an iteration through all processes is simulated, however, while any of the processes is running, the value of currentId should not be changed at all; note that when a process is running we assume that the scheduler does not have access to the CPU, so it should not be able to change the currentId at all. Moreover, if a process is running, we should wait till it releases the CPU and then give CPU to another process. Accordingly, the next value of currentId is the same as the previous one if pcb.lock is true in the next step, meaning that a process is taking the CPU and become active, so we should wait till the process releases the CPU. Also, the next value of currentId is the id of the process in the list so that the Round Robin-like strategy could be simulated, (currentId + 1) mod N, where N is the number of processes existing in the pcb element.

Note that the scheduler currently uses a Round Robin-like strategy, but it can be modified to other strategies. In order to model check an application using different strategies, e.g. FIFO or FILO, for selecting next running process the scheduler should be modified so

[2]In the current strategy, there is no need to pass the i/o pins to the scheduler module, however if another strategy is going to be used the input variables can be passed to the scheduler, so that it can read their values and set the currentId value based on the value read from the input pins.

that setting the next value of currentId simulates that specific strategy.

4. SharedVariable module definition

   (a) For each variable defined in the node scope of the program, and also for each i/o pin defined in the program, a variable should be passed to SharedVariables module as input parameter. SharedVariable is responsible for setting the value for the shared node variables and output pins.

   (b) Set the signature of the SharedVariables element. First, a name should be selected and a set of needed parameters should be set as input arguments. the name of the SharedVariables element will be "node_nodeName_SharedVariables" where nodeName is the name of the node in the source program. Secondly, its input parameters should be set as pcb, event, all the variables defined in the node and are accessible by all the processes, and all input and output pins that are defined in the source file.
   MODULE node_nodeName_SharedVariables(pcb, event, nodeVariable*, outputChannelName*, inputChannelName*);

   (c) Assign values to the variables that are defined in node scope of the program and are actually shared between the processes. Rules regarding the next value of each of the shared variables should be put in ASSIGN block of the module.

      - Set event variable. The event variable should be initialized and modified in the sharedVariables module. We expect event to show all the triggered internal events; therefore, at the beginning of the execution of the program we should initialize it to zero, which means that no event has been triggered. The value should be set based on the size of the event variable, which has been defined in node module, e.g. event: word[2] in node means there are 2 events defined in the program and the event variable is a two-bit word; therefore, it should be initialized by init(event) := 0b2_0;
      - Set node variables, that are the variables shared between processes of a model and could be modified based on the status of the processes. These variables should be defined in the node scope and modified inside sharedVariables module.
      - Set input/output pins. for each input or output port defined in the ArduinOS application, a variable should be considered in the model. The input variables should be set equal to one the other output variables. Therefore, inside the model, the outputs can be assumed as variables shared between the processes of a node, and should be modified based on the programmed logic.

5. PCB module definition

   (a) For each variable defined in the node scope of the program, a variable should be passed to PCB module as input parameter, and pcb should also pass it to all of its inner processes, while instantiating them.

(b) Set the signature of the PCB element. First, a name should be selected and a set of needed parameters should be set as input arguments. the name of the PCB element will be “node_nodeName_PCB” where nodeName is the name of the node in the source program. Secondly, its input parameters should be set as currentId, event, all the variables defined in the node and are accessible by all the processes, and all input pins that are defined in the source file.

MODULE node_nodeName_PCB (currentId, event, nodeVariable*, inputChannelName*)

(c) Define the variables of the PCB module in its DEFINE and VAR blocks as follows:

- Instantiate all the processes defined in root method of the program. PCB is actually simulating the process list that is defined in the root of the program. For every process in root method, an instance of a process should be defined in pcb VAR block. Since the ordering of the processes affects the execution scheduling, we instantiate the processes in the same order they have been defined in root method.

  For each process declared in the program we should have a process element (module) defined in the model (the definition of the process modules will be explained in the next steps; at this level we just assume that we have the modules defined, and we instantiate the processes based on this assumption.)

  Each process should be instantiated using the signature of its corresponding module, and a unique identifier and all input parameters of the pcb should be passed to it. We name each process instance as “process_i” while i is the identifier chosen for the process and should be a unique number from 0 to N-1, where N is the number of processes defined in root/pcb. Accordingly, in pcb, the identifiers chosen for processes starts from 0, and will be incremented for each process. Other than identifier, the pcb parameters including currentId, event, all the variables defined in the node, and all defined input pins should be passed to that module. The name of the module for each process will be set as node_nodeName_ProcessName, where nodeName is the name of the node defined in the source file, and ProcessName is the name of the process given to it when it was being declared in the source program. According to our defined naming rules, for each process_i that is being defined we have the following statement in the VAR block of the PCB module.

  process_i : node_nodeName_ProcessName (i, currentId, event, nodeVariable*, inputChannelName*);

- Define lock variable for PCB. The pcb lock variable which shows if any of the processes are running or all of them are idle should be defined in the DEFINE block. The pcb lock is true if at least one of the processes are locked (the lock variable of the process is true, meaning that the process is running). Therefore the lock can be defined as a rule in DEFINE block so that it is the result of logical OR between all process locks. lock := process_1.lock | process_2.lock | process_3.lock; where process_1, process_2, and process_3 are the name of the processes instantiate in PCB.

6. Process modules definition

   Define process modules and rule variable assignments based on the programmed logic. For each process declared in the program we should have a process element (module) defined in the model. Each state defined in the process may affect on the value of process and node variables. Accordingly, all the variable modifications should be coordinated based on the commands provided in the states of the process. To achieve that, first the process variables should be defined and initialized based on our modeling assumption. Then the logic in the state transition should be translated to value assignments for process variables (represented in the defined process module) and node variables (represented in sharedVariable module).

   (a) Define all process variables needed according to the modeling schema and initialize them.

      - Define process lock so that it is true if the process is in one of the running states, the process is not in wait or ready state, lock := state != wait & state != ready;
      - Define pevents as a n-bit word where n is the maximum number of events and interrupts defined in the program. Respectively, pstate should be defined as an array of size n of which elements can have one of the active states defined in the program. The process is not waiting for any events at the very beginning of the execution, therefore, pevent should be set as 0 (no events), and the value of pstates does not matter since the process is not waiting for any event and does not care about the value stored in pstate.
      - State variable, representing the current state of each process, is defined and the list of its possible values consists of “wait”, “ready” and all defined active states in the program. The state variable should initially be set as “ready” since all processes are ready to get the CPU and be activated.
      - nextState variable, representing the next running state in which the process should be woken up, should be defined with the domain of all defined active states. Initially, nextState represents the start state of the process, which according to our assumptions is the first active state defined in the process.
      - Timer should be defined as an integer, from 0 to the maximum needed delay time programmed. It represents the number of steps a process should wait for a time event. The state in which the process should be woken up is kept in tstate, of which domain includes defined active states. Initially, timer is 0 and the value of tstate

does not matter since the process is not waiting for any time event.

(b) For each active state defined in the program, find the conditions effective on the value of each process and node variable, and calculate the one-line condition for value assignment for the variables. All the commands coded in each state should be executed in only one step in NuSMV model, so that it seems all the variable modifications have happened simultaneously. Therefore, the problem will be changed to translating a procedural piece of code, possibly consisting of conditional and loop statements, into functional assignments for each variable.

   i. Detect the unit blocks of code. Each block is defined as a set of lines in a state definition which can be compressed as a value assignment rules for variables. If the state definition does not contain any loops and all its statements are regular commands or conditional statements, the corresponding block would contain the whole body of state.

      However, if the state contains a loop the state needs to be decomposed to beforeLoop, loop, and afterLoop states to model the corresponding control flows. Since NuSMV does not have any notion of loops, the loop is converted to a series of inner state transitions. To achieve this, state S which contains a while loop in its body should be considered as three individual states S_beforeLoop, S_loop, and S_afterLoop states, so that S_beforeLoop passes the execution flow to S_loop (like a goto(S_loop) statement in DSL), and S_loop conditionally passes the flow to the beginning of S_loop or S_afterLoop. The same approach should be followed recursively for all states until there is no loop in any of the blocks (states). Then each state can be assumed as a unit block which is translatable to variable assignment conditions. Figure B.1 shows the intermediate interpretation of such case. As it is shown, in the interpreted version instead of S we have three active states; therefore the possible values for state, pstate and tstate should be updated respectively.

   ii. Find the updated value for each variable, and the conditional paths resulting to the new values. Each variable may be modified in the block and its new value may be set to constants, its own old value, the value of other variables or a combination of these. For instance, at some point we may have $A = A + B + 2$ where A and B are variables which might have been updated several times in the same block. Therefore, the new value of A depends on the value of A itself and the value of B, and all execution paths that may affect the value of these two variables also affect the A's new value, and should all be considered as part of conditional assignment statements for A. Accordingly, we need an algorithm for extracting the right values considering the original order of statements in the procedural program.

      Detect all possible execution paths of the block. Each simple statement (assign-

```
state S{
  //some computations
  counter = 0;
  while( counter < 5){
    if(x==y){  //some condition
      counter=5;
    }
    //some other computations
    counter = counter+1
  }
  //some computations
  a=0
}









 .
```

Listing B.2: Original While Loop

```
state S_beforeLoop{
  //some computations
  counter = 0;
  goto(S_Loop);
}
state S_Loop{
    if(counter < 5){
    if(x==y){  //some condition
      counter=degree;
    }
    //some other computations
    counter = counter+1
    goto(S_Loop);
  }
  else{
    goto(S_afterLoop);
  }
}
state S_afterLoop{
    //some computations
    a=0
}
```

Listing B.3: Interpreted While Loop

Figure B.1: Handling While Loops Example

ment statements, wait, delay, changes in status of the process) modifies the value of at least one variable, and conditional statements are the ones that branches the execution paths. Assuming that, we can build a tree structure representing the execution paths of the block:

A. Each node keeps the value of all process and node variables, and if any of them has been modified in any branches, the node keeps only its final value.

B. Each edge has a label, representing the conditional statement of that branch, is an expression based on the value of some variables at that point in the execution path.

While building the tree, if the value of a variable in any of the leaves is different from its value in the root of the tree, it means that variable has been modified in the block. Therefore the path (sequence of branching labels) from root to that leaf represents the condition of the target NuSMV variable assignment statements, and the corresponding value of the variable is the one showed in the leaf.

iii. Given the condition-value pairs for each variable, the assignment statements are placed into the corresponding module. The process variables assignments (pevent, pstate, timer, timeState) should be done in the body of the process module, whereas the assignment statements for variables shared between processes (node variables, input and output variables, events) should be put in sharedVariable module of the node.

# Appendix C

# Translating ArduinOS Programs Into Arduino Executable Programs

Recall that the defined ArduinOS DSL was designed so that we limit the domain of programs can be written in ArduinOS acceptable format. In fact, Arduino language, which is actually based on C++, is more capable than our DSL, and our defined language is just a subset of what can be written in C++, or the language used by Arduino devices. Besides, C++ has the same procedural line-by-line execution of the program as our language; therefore, there is no need for complex in-line function extractions in the translation procedure.

Although the logic programmed in our defined language can easily be translated to C++, in order to cover the expected underlying kernel layer logics, we should design a fixed structure for the intended code in Arduino language. By kernel layer logics, we mean all the node does for scheduling the processes, e.g. iterating through the instantiated processes, triggering and handling events, keeping track of the time for those processes waiting for a time event, and etc. Since the underlying kernel logic of the program has been assumed the same as PicOS kernel layer, we need to design a structure covering the same notions of PCB, Process, events, and process states. In the remainder of this section we provide the structure used in Arduino programs and explain the steps needed for translating a program written in ArduinOS DSL to the structured C++ one that is acceptable by Arduino devices.

Our applications, programmed in the DSL, have been written based on PicOS approach in managing processes. Therefore the structure of the destination C++ program should be compatible with PicOS kernel design. According to PicOS logics, and also based on assumptions about how the program should work, the node needs to have a list of processes each of which contains a partial behavior of the node. In order to keep our element naming the same between the NuSMV model, PicOS application and the C++ Arduino programs, we call the list of processes as PCB (Process Control Block) since it is the element facilitating management and controlling the processes. Accordingly, the node mainly has a pcb and some variables, and can iterate over processes of PCB and run them one at a time.

Each element in PCB should be of type Process, and should be runnable, i.e. should have a run method that contains the application layer logics defined by the user. Process is an abstract class that provides a virtual (a.k.a. abstract) method called run. Each class that is a subclass of Process has to implement the run method based on what behavior it wants to express. Each Process instance may be in wait, ready or active state. While waiting, the process may be waiting for a list of events, and after occurrence of any of the interesting events it is expected to wake up at a specific event, Therefore for each interesting event there should be an state for event in which it wants to wake up; if the event occurs, the related state should be set as the nextState in which the process will be activated. Besides, each process has its timer that is synced with the node timer, and can trigger the timeEvent. Listing C.1 shows the Process structure.

```
class Process {
public:
  int timer ;
  int timeState;
  myVector <int, MAX_EVENTS> waitingStates;
  myVector <int, MAX_EVENTS> waitingEvents;
  int nextState;
  int state;
  virtual void run(int &orientation, int &i_right, int &i_left,
      int &o_right, int &o_left) =0;
  Process(){
    timer =0 ;
    state =ready;
  }
};
```

Listing C.1: Arduino Abstract Process Class

Other than inherited properties, the subclasses of Process should have an activeState. The activeState can be one of the states defined in the original application, and the transaction between these active states should be programmed in the run method. In the run method, the process should check if it is ready to be activated or not and if any of the interesting events had been occurred; if yes it goes to one the possible active states and performs the programmed behavior of that state, then it releases the CPU and goes to wait state again. The generic structure of a concrete process has been illustrated in Listing C.2. When the process is running in any of the active states, it may want to invoke methods for triggering an event and announcing it to other processes, or modify variables to set new values to them or wait for events[1].

```
class p_concreate : public Process {
public:
  enum activeState { s1, s2, s3 };
  activeState activestate;

  p_concreate()
```

[1]Details on how each of these functionalities can be performed and translated from the original program will be provided later in this section

```
  : Process(){
    nextState = s_init;
  }

  void run(int &nodeVariable, int &node_input, int &node_output){
    if (state==ready) {
      activestate = (activeState) nextState;
      state = active;
    }
    else return;
    switch (activestate) {
      case s1:
          //do some computation, wait for events, or trigger some
        break;
      case s2:
          //...
        break;
      case s1:
          //...
        break;
      default :
        ;
    }
    state = wait;
  }

};
```

Listing C.2: Arduino Concrete Process Class

Knowing about the concept of each element and method, here we focus on the step-by-step translation procedure of an application in ArduinOS format to a C++ program. However, instead of translating each command of the source code to the related logics in the C++, we provide the template in which the program should be written and how to fill the template based on the source file. This approach will be more straight forward for manual model extraction and it is also more understandable. Following is the list of all the steps necessary for building the C++ program out of the AduinOS application.

1. Create Abstract class Process. The original application is the definition of a set of processes, which are defined and then instantiated in the root method. Since we want to model the processes we extract all of common properties between the processes and what they need to execute. Therefore, we need a fixed structure to keep these common properties, and we call it as Process. Process is an abstract class, meaning that we cannot have an object of its kind, but it can be the superclass of other classes. We define the virtual run method for process, i.e., it has no body but it is forced to be implemented in all subclasses of Process. Besides, Process contains all properties common between runnable processes on a node which are as followings:

- timer is an integer variable of which value shows the number of steps remaining for a time event being triggered for the process. If the timer is bigger than zero it means that the process is waiting for a time event. At each step of running the program, the timer decreases if it is bigger than zero, and when it reaches zero and the process is still waiting for the time event, the process will become ready to run.
- timeState is the variable storing the next state the process should wake up if it realizes a time event being triggered.
- waitingEvents is the list of all events for which the process is waiting. If a process is in wait state it may be waiting for some events to happen. If any of those interesting events happens, the process becomes ready to run and it will not keep waiting for events anymore. All the interesting events for which the process is waiting should be added to waitingEvents and whenever the process becomes ready, waitingEvents will be cleared, meaning that the process is not waiting for any event anymore.
- waitingStates is the list of all states in which the process wants to wake up if any of the interesting events trigger. When a process sets an event as an interesting one and adds it to the waitingEvents, it should also select one of its activeStates and add it to waitingStates, so that the node know in which state the process should be invoked if one of the interesting events occur.
- nextState is the state in which the process will be activated. Anytime the process realizes the occurrence of an interesting event or the time event, it should set the nextState value according to the waitingStates or timeState value, depending on the type of the event that has been realized.
- state is an integer showing if the process is waiting, ready to run, or active at any time.

All these considered, the application needs to have Process class as it is defined in Listing C.1.

2. Define pcb as a list of all process instances in the node. In order to keep track of all processes running in the node, we need to have a list of them stored as one of the variables of the node. In that case, we can iterate over the processes, schedule them for being executed, call their run method, and notify them about an event being triggered, and etc.

   vector <Process*> pcb;

3. Define the trigger method that receives an event as the input and is capable of triggering events and announcing its occurrence to all the processes in the node. One of the actions that each process may perform is to trigger an event; therefore, trigger should be a method invocable by all processes defined in the program. On the other hand, any time an event is triggered we want all the processes to be informed about that and check if any of them is expecting the event. As one of the interesting events occurs, the process' state should be changed from wait to ready. These are the functionalities that we expect the trigger method to provide; receiving

an event, it should go over all the processes in pcb and check if the event is interesting for the process or not, if yes it should make the process ready, set its nextState to the waitingState index that points to the related state to the event that just occurred, flush its waitingEvents and waitingStates, and reset the timer so that the process is not waiting for any other events. The implementation of trigger method is the same for all programs and it is shown in Listing C.3.

```
void trigger (int event){
  for (int i=0;i<pcb.size();i++){
    for (int i=0;i<pcb[i]->waitingEvents.size();i++){
      if(pcb[i]->waitingEvents[i] == event){
        pcb[i]->nextState = pcb[i]->waitingStates[i];
        pcb[i]->waitingEvents.clear();
        pcb[i]->waitingStates.clear();
        pcb[i]->timer =0;
        pcb[i]->state = ready;
      }
    }
  }
}
```

Listing C.3: Arduino Trigger Method

4. Define the globalTimertick method responsible for modifying processes' timers. Pcb processes may be waiting for a time event which means they should wait for a specific amount of time till they can be woken up by a time event. Besides, the notion of time for all the processes of a node should be the same, and close to what we consider as time in NuSMV model, i.e. each tick of the clock is one step in state transitions of the variables. Accordingly, a global time counter is needed to synchronize all the timers of the processes and modify their values. Therefore, the responsibility of globalTimertick is to go over all processes in the pcb, and if they have a timer bigger than zero (i.e. they are waiting for a time event) it should decrease their timer value by one, meaning that they are one step closer to realizing the timer event. Moreover, if the value of a process' timer reaches zero it means that the process can realize the timer event now and become ready; therefore, the method should make the process ready, set its nextState to timeState, and flush its waitingEvents, waitingStates and reset the timer so that the process is not waiting for any other events. The implementation of globalTimertick method is the same for all programs and it is shown in Listing C.4.

```
void globalTimertick(){
  for (int i=0;i<pcb.size();i++){
    if(pcb[i]->timer != 0){
      pcb[i]->timer = pcb[i]->timer - 1;
      if(pcb[i]->timer == 0){
        pcb[i]->nextState = pcb[i]->timeState;
        pcb[i]->waitingEvents.clear();
        pcb[i]->waitingStates.clear();
        pcb[i]->state = ready;
      }
    }
    }
}
```

Listing C.4: Arduino Timer Tick Method

5. Implement the basis of application in the main method of the C++ program.

    (a) Setup Arduino and set the physical pins as the input and output pins used in the program. The resulted C++ program should be executable on Arduino devices; therefore, we need to invoke some initiating methods for setting up the environment for an Arduino application, this can be possible by invoking *init()* method. Moreover, we may want to observe the behavior of the Arduino device by connecting some LEDs to its pins, or make some channels for them to communicate with other nodes in the network. The physical pins should be set as input/output pins at the beginning of the program. As a thumb rule we can consider the number of input/output variables defined in the original program to find out how many pins we need in the C++ code, e.g., for each input bit defined in the *.node* file we need a pin to be set as INPUT, and respectively for every output bit defined we need a pin to be set as OUTPUT. Besides, the list of processes should be set before the main looping execution. Therefore, for each process defined, we need to instantiate a new object and add it to the PCB. Note that, later in the program for scheduling the processes we should iterate over the PCB and select the next process to run; therefore, in order to have the same scheduling in all models, the order of processes in the pcb should be the same as their order in the original ArduinOS program.

    (b) Define node variables and the input/output interfaces. For each variable defined in the node scope of the .node file, we need to define a variable in cpp file. The translation is straight forward in this case and the variable definitions can be directly copied from the original program. We need input and output intermediate buffers to keep the variables read from input ports or are intended to be written on the output ports. Before activating any process, the input variables are read from input channels (pins), their value will not be changed until the next update. After the execution of any process is done, since the output variables may be changes by processes during their execution, their new values should be written to the output channels. Accordingly, for every bit of input (output) one bit variable is needed as a node variable. For the sake on simplicity, we can assume that for each connection we can combine all bits of inputs (outputs) to only one input (output) variable, meaning that instead of receiving (sending) parallel bits we can have one multi-bit packet. In that case, some bit modifications are needed for assigning the composed variable to parallel ports (pins).

    (c) Continues loop execution of the processes should be implemented. The loop method is actually a forever loop which is executed over and over again while the node is plugged. At each call, all processes in the application should be scheduled for execution. Accordingly, it contains the main process scheduler.

The scheduler should go over the processes in the PCB, and call the run method for each process. Since every process in the node may need the received inputs, the value of the input variable should be updated before running each process. Respectively, each process may affect the value of the output variables; therefore, the output ports are needed to be updated based on the new values of output variables. Accordingly, the scheduler iterates through the processes, and for each process 1) updates the input node variable, 2) runs the process, and 3) updates the output pins based on output node variables.

6. Define an individual class for each process defined in the program. The defined classes for a process should be a subclass of Process class, and should be named according to its corresponding name in the original *.node* file. Then the internal structure of that process should be translated as follows:

   (a) Define the list of all possible states the process can get, and define the variable keeping that state for the process, activestate. This can be possible by defining an enum including all running states defined in the original *.node* file.

   (b) Set the start state of the process in its constructor. The class needs to have a public constructor that first calls the constructor of its superclass (Process) and then sets the start state of the process. The start state is the initial value of the nextState variable. Assuming the first state defined in the process as the start state, the value of nextState should be set to the first running state defined in each process.

   (c) Implement the run method for the process, containing the main behavior of the process. The interface of run method is inherited from Process. The signature of the method is the same as the one defined in Process class and depends on the inputs and outputs a node has, and it also should have the access to all variables defined for the node. In order to implement the *run* method for each process the following steps should be completed:

      i. Activate the process if it is ready to run. The first step in execution of each process is to check if the process is ready (runnable) or not. If the process is ready, it should change its state to active (running) and update the value to one of its activestate according to the expected nextState. Otherwise, if the process is not ready it means that it is still waiting for some events to occur; therefore, the method should be terminated.

      ii. Transform the state-based structure of the program into a switch statement on process active state. Therefore, instead of having states, here we have cases. At first, a switch statement should be defined based on process activestate. Then, for each state in the process a case should be created in the switch statement and the whole body of the state definition followed by a "break" statement should be copied to the corresponding case. Therefore, case s_0 contains the exact body of the state s_0 definition from the original program. Next the following steps should be done to translate some DSL specific statements into the C++ Arduins executable ones.

- For each command wait(EVENT, STATE) in the .node program, 1) add EVENT to the list of events that the process is waiting for waitingEvents.add(EVENT); and 2) add STATE to the list of states in which the process should be woken up if the corresponding event occures, waitingStates.add(STATE);
- For each command delay(TIME, STATE) in the .node program, 1) set timer to TIME so that the process waits for at most that amount till next execution (timer = TIME); and 2) set the timeState to the STATE in which the process should be woken up after the timer goes off (timeState = STATE).
- If there are any command goto(STATE) in any routines, put a label, say L_SWITCH, at the beginning of the switch-case statement, then, instead of goto(STATE) update the active state to the STATE (activestate = (activeState) STATE) and then force the routine to jump to the beginning of the switch-case statement (goto L_SWITCH).
- Set the process state to wait at the end of all cases. Before ending each case by breaking its normal flow, the process state should be set to wait, which means the process is releasing the CPU.