© 2008

JOSEPH CWIKLA

ALL RIGHTS RESERVED

SPECIFYING, IMPLEMENTING AND VERIFYING LAYERED NETWORK
PROTOCOLS

A Thesis

Presented to

The Graduate Faculty of The University of Akron

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

Joseph Cwikla

August, 2008

SPECIFYING, IMPLEMENTING AND VERIFYING LAYERED NETWORK
PROTOCOLS

Joseph Cwikla

Thesis

Approved:

Accepted:

Advisor
Dr. Kathy J. Liszka

Dean of the College
Dr. Ronald F. Levant

Faculty Reader
Dr. Timothy W. O'Neil

Dean of the Graduate School
Dr. George R. Newkome

Faculty Reader
Dr. Yingcai Xiao

Date

Department Chair
Dr. Wolfgang Pelz

ABSTRACT

As computing power increases, software is developed to make use of the increased capacity. A transition is currently in progress as the growth of higher level scripting languages enables a larger number of programmers to develop programs in larger and more diverse domains. The world of software program development is becoming readily accessible to a larger audience.

Concurrently, digital communication has revolutionized the way that people interact. Protocols are established to enable new types of communication. Successful protocols are built in layers; each layer makes use of the one below it and providing services to the layer directly above it. As new uses are discovered for digital communications, new protocols will need to be developed to support them.

Simulations are useful when communication protocols are developed. The simulation can be specified, implemented and its runtime behavior verified against the specification.

This thesis proposes and demonstrates that there exists a set of software development tools that are readily accessible and end to end to address the problem of specifying, implementing and verifying layered communication protocol simulations.

TABLE OF CONTENTS

CHAPTER	Page
I. INTRODUCTION	1
II. TOOLS	5
2.1 Specification Tools	6
2.2 Implementation Tools	21
2.3 Verification Tools	25
2.4 Example	29
III. MODEL	36
3.1 The Wireless Medium Layer	37
3.2 The Physical Medium Dependency Layer	38
3.3 The Wireless Channel Layer	38
3.4 The Physical Layer Convergence Procedure	39
IV. SPECIFICATION	41
4.1 Wireless Medium Specification	41
4.2 Physical Medium Dependency Specification	45
4.3 Wireless Channel Specification	47
4.4 Physical Layer Convergence Procedure Specification	48
V. IMPLEMENTATION	51
5.1 Wireless Medium Implementation	52

5.2 Physical Medium Dependency Implementation	61
5.3 Wireless Channel Implementation.....	69
5.4 Physical Layer Convergence Procedure Implementation.....	74
VI. DISCUSSION.....	84
BIBLIOGRAPHY.....	87
APPENDICES	89
APPENDIX A WIRELESS MEDIUM VERIFICATION CODE	90
APPENDIX B PHYSICAL MEDIUM DEPENDENCY VERIFICATION CODE.....	99
APPENDIX C WIRELESS CHANNEL VERIFICATION CODE	106
APPENDIX D PHYSICAL LAYER CONVERGENCE PROCEDURE VERIFICATION CODE.....	112

CHAPTER I

INTRODUCTION

Communication is ubiquitous and vital to our lives. It is also difficult to manage correctly. Protocols are established and when followed permit some subset of possible messages to be transmitted and hopefully received. When introduced to someone that we don't know we smile, possibly shake hands and say how glad we are to know that person. A simple protocol but what set of messages can it convey? Not a large set certainly. As the world leans more and more on electronic communication the set of messages that need to be communicated across computer networks is growing. The protocols will need to grow as well. But what electronic protocols need to be created? What will they accomplish? Development of communication protocols is complicated and risky.

One way to mitigate the risk is to run simulations of the proposed candidate protocols. This eliminates the need for an actual physical system. Rather, a model of the system is employed. For the simulations to be meaningful the model must be precise, even mathematical [1]. Once a precise model is specified, however, simulation provides tangible benefits over working with real world systems [2].

- Time can be modeled. Working with real systems means that real time limits the investigation. In a simulation, time itself can be modeled and expanded or compressed.
- Sources of variation can be controlled. These sources of variation must be explicitly defined and it's possible to limit these to the sources of interest.

- In real world systems measurement error is inevitable. In a simulation this is not a concern since real world measurements are not being made. It is possible to stop and review a simulation. All components in the simulation are frozen and a snapshot of global state can be obtained and restored at any time. The value of simulation is indisputable when developing new network protocols.

General purpose protocols need to be layered. The OSI model [3] defines seven layers, while TCP/IP [4] defines five. Where would the internet be today if TCP were the only socket interface available for network programmers? Certainly it's the most used transport by programs that communicate over the internet but the expense of connection setup and tear down sometimes makes it a sub-optimal choice, especially when a reliable transport isn't necessary. UDP serves a different crowd.

So, not only is it necessary to simulate, but it's necessary to simulate at different layers, which means that it is necessary to reason about those layers independently and then be able to combine them afterwards. More generally, it is advantageous to reason about a *system* at various levels of abstraction in a way that permits conclusions to be drawn at each level. Even more advantageous is being able to follow that up with a composition of those components, drawing conclusions on the composed system based on the conclusions about the components.

Motivation for this work grew out of an experience with the use of a proprietary product to develop a network simulation for wireless networks. The product was fine, well suited for the task at hand. Issues came up however with license renewals and the inability to share the work that was being done with anyone who did not have a license for the product. Also, it required a "try it and see" approach. There was no discernable way to specify ahead of time within the tool set what the system was supposed to do. Rather, it was necessary to implement it first and see what it did.

The concepts of *readily accessible*, and *end to end* tools permeates this effort. The approach being advocated is that the problem be understood and rigorously specified, a solution attempted and then the results verified against expectations. So the challenge is to find the right set of tools for the specification, implementation and verification phases; a set of tools that meet the *readily accessible* and *end to end* goal.

The specification tools while being rigorous need to be understandable without special training. They need to be simple yet powerful enough to specify complex systems. They must provide facilities for composition so that behavior of network layers can be specified individually and yet connected together to specify a complete protocol stack. The tools for specification should generate output that can be consumed and utilized by the implementation phase of the development. That is, there should be a link between the specification and implementation phases.

The implementation tools must also be simple yet powerful. They must be capable of implementing complex, composed system specifications while enforcing a separation of concerns that expresses that specification more simply as an executable composition of its components. The implementations must generate output that can feed the verification phase. Thus, the tools must be amenable to capturing runtime behavior that can be determined to have met the specification. The implementation tools must easy to learn and readily accessible.

The verification tools must be able to consume the output of the simulations and demonstrate that the specifications have been satisfied, or not satisfied. Thus, there must be linkage between the verification tools and both the specification and execution of the simulation. verification.

The rest of this thesis attempts to show such a tool set. It identifies specific tools that are appropriate for each phase. It explains how the development can flow within that set of tools

from specification through implementation and verification. Each phase is shown to be linked to the next in the methodology described herein.

CHAPTER II

TOOLS

A programmer has an idea. He uses a text editor to express that idea precisely in source code. He uses a compiler to transform that source code expression into an object code expression of the idea. He uses a linker to transform the object code expression into an executable expression of his idea. The executable expression is run on a computer and shown to satisfy the idea. Text editor, compiler, linker, computer: a powerful tool set. The tool set is linked, each tool takes the output of the previous one until the final result is achieved. Text editors, compilers, linkers and computers are all, and have been for some time, readily available to any programmer. This simple, readily available, end to end linked tool set has provided development pipeline for many amazing products.

The value of an integrated methodology for specifying and verifying network protocols has long been recognized [5] and research continues for today's protocols [6]. The methodology advocated in this thesis defines an analogous tool set to that described above for the composition of layered network simulations. Beginning with a specification, a precise description of an idea is created. This initial specification is expressed in terms of allowable external behavior; it is a *behavioral* specification. An *implementation* specification can then be developed that is intended to satisfy the behavioral specification. The implementation specification defines precisely a component's behavior, both internal and external, and can serve as input for translation into an executable version of that component. Implementation tools can be identified that

facilitate that translation. Finally, executions of the components must be run and verified against that behavioral specification to ensure correctness. It is the goal of this paper to identify a tool set and a method that achieves these goals for the development of layered network simulations.

2.1 Specification Tools

It's been noted [7] [8] that an implementation of a system is correct if its actual behavior is allowed by its specification. The specification tools must enable precise descriptions of allowable behavior. They must also be usable without special skills or training. Additionally, given a target of layered network simulations, the tools must provide a way to model a shared medium at the lowest layer. The specification tools [7] described below are:

Trace Properties

Trace properties are used to describe observable, that is external, component behavior. Using simple set theoretic reasoning trace properties can precisely and simply define behavior that can be externally verified. Trace properties are a *behavioral specification*.

I/O Automata

I/O Automata precisely define the behavior, both internal and externally observable, of a component. A component defined with an external interface that is compatible with a trace property can be designed to produce only observable behavior that is allowed by the trace property. Only a basic understanding of set theory is needed to utilize I/O Automata. I/O Automata are an *implementation specification*.

Shared Variable Types

Shared variable types are a formalism used to restrict the set of operations that may be performed on a variable shared by multiple processes. They are needed here to specify the behavior of the physical medium through which communication signals travel. Two shared variable types are utilized, a read-write type and a read-modify write type. As with all of the other specification tools, shared variable types are understood using basic set theory. Shared variable types provide both *behavioral* and *implementation* specifications.

I/O Automata

I/O Automata are simple state machines that can be used to model asynchronous components of a system. The basic model has no notion of time and thereby provides a very general model for the interaction between independently acting entities. State transitions of an I/O Automata are associated with named actions which supports a pre-condition, effect style of implementation specification thereby providing a very code-like expression of each action's functionality.

An I/O Automata A consists of five components:

- The signature of A , $sig(A)$, is the set of named actions, *acts*, associated with A .

Each action is either an *input*, *output*, or *internal* action.

$$sig(A) = input(A) \cup output(A) \cup internal(A)$$

- The set of *input* actions and the set of *output* actions provide the *external interface* of A .

$$external(A) = input(A) \cup output(A)$$

- The set of *internal* actions and the set of *output* actions provide the *locally controlled actions* of A .

$$\text{local}(A) = \text{internal}(A) \cup \text{output}(A)$$

- The states of A , $\text{states}(A)$, are described by a set of state variables. State variables can be updated only by an *action* of A .
- The start states of A , $\text{start}(A)$, is a non-empty subset of $\text{states}(A)$.
- The state transition relation of A , $\text{trans}(A)$, defines the relation between actions and state transitions. $\text{trans}(A) \subseteq \text{states}(A) \times \text{acts}(A) \times \text{states}(A)$
 - An element (s, π, s') of $\text{trans}(A)$ is called a step, or transition of A .
 - Each step is classified as an *input*, *output* or *internal* step depending on whether π is an *input*, *output* or *internal* action.
 - If (s, π, s') is an element of $\text{trans}(A)$ then the *action* π is said to be enabled in s .
 - All input actions π must be enabled in all states s . An I/O Automata places no restrictions on its execution environment.
- The task partition of A , $\text{tasks}(A)$, partitions the *actions* that A has control of into separate, independent tasks.

Executions and Traces of Automata

An execution of I/O Automaton A , $\text{execution}(A)$, is a sequence of alternating states and actions of A beginning with an element from $\text{start}(A)$ and ending with an element from $\text{states}(A)$.

$$\text{execution}(A) = s_0, \pi_1, s_1, \pi_2, \dots, \pi_r, s_r \dots$$

The trace of an execution of A , $trace(A)$, is the subsequence of the execution consisting of elements from $external(A)$. $trace(A)$ describes A 's externally observable behavior during the execution. Trace Properties

A trace property, P , consists of two components:

- A trace signature, $sig(P)$, which consists of external actions.
- A set of sequences, $traces(P)$, of actions from $sig(P)$ which defines the ordering of actions allowed by the property.

A trace property specifies an interface and the allowable behavior at that interface.

An I/O Automata A can be said to satisfy a trace property P , if $external(A) = sig(P)$ and $traces(A) \subseteq traces(P)$. That is, if the signature of P is the same as external signature of A and the set of all the traces that A can produce is contained in the set of traces allowed by $traces(P)$ then A satisfies P . In this case A can be said to be linked to P .

A Simple Example Define a trace property, *TerraProperty*, such that:

- $sig(TerraProperty) = \{sunlight(b), day, night\}$ where b is a boolean
- $traces(TerraProperty)$ is the set of sequences of actions from $sig(TerraProperty)$ such that:

1. *day* and *night* alternate, beginning with *day*.
2. After each *day* and before the next *night* there must be at least one *sunlight(false)*.
3. The first instance of *day* must be preceded by a *sunlight(true)*.
4. After each *night* there must be at least one *sunlight(true)* before the next day.

Now define an I/O Automata, *Terra* whose function is to recognize when the sun is shining and transition between day and night accordingly.

- Terra Signature
 - Input Actions = $\{\text{sunlight}(b)\}$ where b is a boolean
 - Output Actions = $\{\text{day, night}\}$
- Terra States = $\{\text{sun, lastsun, newsun}\}$, all boolean and initially false.
- Terra Task Partition = $\{\text{day, night}\}$
- Terra Transitions
 - *sunlight* pre-conditions: None
 - *sunlight* effects: $\text{lastsun} := \text{sun}, \text{sun} := b, \text{newsun} := \text{true}$
 - *day* pre-conditions: $\text{newsun}=\text{true}, \text{lastsun}=\text{false}, \text{sun}=\text{true}$
 - *day* effects: $\text{newsun} := \text{false}$
 - *night* pre-conditions: $\text{newsun}=\text{true}, \text{lastsun}=\text{true}, \text{sun}=\text{false}$
 - *night* effects: $\text{newsun} := \text{false}$

Some possible executions of *Terra*, with the *state* entries listed in parentheses and corresponding to $(\text{sun}, \text{lastsun}, \text{newsun})$, are:

- $(\text{false}, \text{false}, \text{false}), \text{sunlight}(\text{false}), (\text{false}, \text{false}, \text{true}), \text{sunlight}(\text{false}), (\text{false}, \text{false}, \text{true})$
- $(\text{false}, \text{false}, \text{false}), \text{sunlight}(\text{true}), (\text{true}, \text{false}, \text{true}), \text{day}, (\text{true}, \text{false}, \text{false}), \text{sunlight}(\text{true}), (\text{true}, \text{true}, \text{true})$
- $(\text{false}, \text{false}, \text{false}), \text{sunlight}(\text{true}), (\text{true}, \text{false}, \text{true}), \text{day}, (\text{true}, \text{false}, \text{false}), \text{sunlight}(\text{false}), (\text{false}, \text{true}, \text{true}), \text{night}, (\text{false}, \text{true}, \text{false})$

Does the I/O Automata *Terra* satisfy the trace property *TerraProperty*? It does if $external(Terra) = sig(TerraProperty)$ and $traces(Terra) \subseteq traces(TerraProperty)$.

$$external(Terra) = \{sunlight, day, night\}$$

$$sig(TerraProperty) = \{sunlight, day, night\}$$

TerraProperty.1

day must precede *night* because the state variable *lastsun* is initially false. A precondition for *night* is that *lastsun* is true and the only way that *lastsun* can become true is by a *day* action. Therefore *night* cannot be generated before *day*. Thus, *Terra* cannot generate a sequence of external actions that violate this specification.

TerraProperty.2

A precondition for *day* is that the state variable *sun* be true. A precondition for *night* is that *sun* be false. The only action that updates *sun* is *sunlight(b)*. So, after a *day* action a *sunlight(false)* must occur to establish the precondition for *night*. Thus, *Terra* cannot generate a sequence of external actions that violate this specification.

TerraProperty.3

day cannot be generated until the state variable *sun* is true. *sun* is initially false and since the only action that updates *sun* is *sunlight(b)* there must be a *sunlight(true)* before the first *day* action can occur. Thus, *Terra* cannot generate a sequence of external actions that violate this specification.

TerraProperty.4

This specification is a corollary to *TerraProperty*.2 Therefore, the traces of all executions of *Terra* are permitted by *TerraProperty*. That is, $\text{traces}(Terra) \subseteq \text{traces}(TerraProperty)$ so *Terra* does in fact satisfy *TerraProperty*. The implementation specification *Terra* is linked to the behavioral specification *TerraProperty*. This simple example has demonstrated how a specification can be written using trace properties and how an I/O Automata can be developed that satisfies, or implements, the specification. The trace property and the I/O Automata are linked. Note how the transitions of *Terra* were written in a *pre-condition* and *effect* style as previously indicated. This is very similar to how an implementation in a coding language could be accomplished. This style of implementation specification facilitates a simple translation to an executable implementation.

The next section explains how this approach applies not only to a single specification and I/O Automata but to compositions of them as well.

Composition of Trace Properties and I/O Automata

An I/O Automata that satisfies a trace property provides an implementation of a specification. For implementations of interest however, a single trace property and automaton are usually insufficient. Systems of interest often must be composed of independent components. Lynch [7] describes composition operations for trace properties and I/O Automata. The composition of trace properties results in a composed trace property while the composition of I/O Automata results in a composed I/O Automata.

The composition of both trace properties and I/O Automata require the notion of *signature compatibility*. For a collection of signatures, $\{S_i\}_{i \in I}$, to be compatible then for all $i, j \in I, i \neq j$ the following must be true:

1. $\text{internal}(S_i) \cap \text{actions}(S_j) = \emptyset$ That is, an action that is an internal action in any of the components cannot be an action of any kind in another component.
2. $\text{output}(S_i) \cap \text{output}(S_j) = \emptyset$ That is, an output action in any component cannot also be an output action in any other component.

These two restrictions ensure that locally controlled actions are limited to a single component in the compositions. A collection of trace properties or a collection of I/O Automata are *compatible* and may be composed if their signatures are compatible.

The composition S of a compatible collection of signatures is defined as:

- $\text{output}(S) = \bigcup_{i \in I} \text{output}(S_i)$
- $\text{internal}(S) = \bigcup_{i \in I} \text{internal}(S_i)$
- $\text{input}(S) = \bigcup_{i \in I} \text{input}(S_i) - \bigcup_{i \in I} \text{output}(S_i)$

Note that an action that is an input action in one signature and output action in another signature becomes an output action in the composition thereby becoming a locally controlled action of the composition and still available for communication with other components in further composition.

Trace properties may be composed if their signatures are compatible.

Composition of trace properties results in a trace property. That is, it results in a signature and an allowable set of sequences of actions from that signature. The composed trace property P is defined as:

- $\text{sig}(P) = \text{composition}(\{ \text{sig}(P_i) \}_{i \in I})$. The composition of signatures previously defined.
- A set of sequences, β , of actions from $\text{sig}(P)$ which defines the ordering of actions allowed by the property such that the projection of β onto actions(P_i), $(\beta \mid \text{actions}(P_i))$, results in a sequence allowed by P_i .

$$\forall i \in I, (\beta \mid \text{actions}(P_i)) \in \text{traces}(P_i)$$

Thus, it is possible to pull from the trace of a composed system the traces of the individual components. This allows examination of the behavior of the individual components independently from the trace of a composed system execution. This characteristic of composed trace properties will be explored further later on.

I/O Automata may be composed if their signatures are compatible. Composition of I/O Automata results in an I/O Automata. A composed I/O Automata A is defined as:

- $\text{sig}(A) = \text{composition}(\{ \text{sig}(A_i) \}_{i \in I})$. The composition of signatures previously defined.
- $\text{states}(A) = \text{composition}(\{ \text{states}(A_i) \}_{i \in I})$. The states of the composition is the cartesian product of the component states.
- $\text{start}(A) = \text{composition}(\{ \text{start}(A_i) \}_{i \in I})$. The start states of the composition is the cartesian product of the component start states.
- $\text{trans}(A)$ consists of the set of tuples (s_i, π, s_i) such that, $\forall i \in I$, if $\pi \in \text{actions}(A_i)$ then the tuple $\in \text{trans}(A_i)$. If $(s_i, \pi, s_i) \in \text{trans}(A_i)$ then $s_i = s_i$. That is, steps, and their corresponding updates to the automaton's state happen component wise.

Recall that when I/O Automata are composed, the actions that are an input action of one component and an output action of another component become output actions of the composition. The composition turns the two independently defined actions and combines them into a single action. This requires that when the composition occurs that output component action and the input component action occur atomically. If further communication via such an action isn't desired then it is possible to *hide* this action, making it an internal action of the composition.

Since the composition of trace properties results in a trace property and the composition of I/O Automata results in an I/O Automata, it is possible to apply compositional reasoning to a system built from components. Importantly, reasoning about individual trace properties and I/O Automata can lead to conclusions about the composition. This means that understanding larger, more complex, composed systems by can be obtained by examining their simpler components individually.

Example Composition

Define a trace property, *SolProperty* such that:

- $\text{sig}(\text{SolProperty}) = \{\text{sunlight}(b)\}$ where b is a boolean
- $\text{traces}(\text{SolProperty})$ is the set of all possible sequences of actions from $\text{sig}(\text{SolProperty})$

If $\text{sunlight}(b)$ is not an output action in both SolProperty and TerraProperty then their signatures are compatible and the two trace properties may be composed. Now define an I/O Automata, Sol whose job it is to output $\text{sunlight}(\text{true})$ half the time and $\text{sunlight}(\text{false})$ half the time.

- Sol Signature

- Output Actions = $\{\text{sunlight}(b)\}$ where b is a boolean

- Internal Actions = $\{\text{incrementDegrees}\}$

- Sol States = $\{\text{integer degrees} = 0, \text{boolean shining} = \text{true}\}$

- Sol Task Partition = $\{\{\text{sunlight}(b)\}, \{\text{incrementDegrees}\}\}$

- Sol Transitions

- $\text{sunlight}(b)$ pre-conditions: $b = \text{shining}$

- $\text{sunlight}(b)$ effects: None

- incrementDegrees pre-conditions: None

- incrementDegrees effects: $\text{degrees} := \text{degrees} + 1$, if $\text{degrees} \bmod 360 < 180$

then $\text{shining} := \text{true}$ else $\text{shining} := \text{false}$

Does the I/O Automata Sol satisfy the trace property SolProperty ? It does if

$\text{external}(\text{Sol}) = \text{sig}(\text{SolProperty})$ and $\text{traces}(\text{Sol}) \subseteq \text{traces}(\text{SolProperty})$.

$\text{external}(\text{Sol}) = \{\text{sunlight}\}$ $\text{sig}(\text{SolProperty}) = \{\text{sunlight}\}$

SolProperty.¹ There is no restriction placed on the sequences of *sunlight(b)* by *SolProperty* so clearly:

$$\text{traces}(\text{Sol}) \subseteq \text{traces}(\text{SolProperty})$$

Some possible executions of *Sol*, with the state entries listed in parentheses and corresponding to (*degrees*, *shining*), are:

- (0, true), *sunlight(true)*, (0, true), ...
- (0, true), *incrementDegrees*, (1, true), *sunlight(true)*, (1, true), *sunlight(true)*, (1, true), ...
- (0, true), *incrementDegrees*, (1, true), ..., *incrementDegrees*, (181, false), *sunlight(false)*, (181, false), ...

Therefore *Sol* satisfies *SolProperty*.

In the two I/O Automata *Terra* and *Sol*, *sunlight* is an external action. But in *Terra* it is an input action whereas in *Sol* it is an output action. Thus the corresponding trace properties, *TerraProperty* and *SolProperty* may be composed. Also, the signatures of *Terra* and *Sol* are compatible so the two I/O Automata may be composed as well.

The composition of the trace properties *SolProperty* and *TerraProperty* results in a trace property *SystemProperty* such that:

- $\text{sig}(\text{SystemProperty}) = \{\text{sunlight}(b), \text{day}, \text{night}\}$ where *b* is a boolean
- $\text{traces}(\text{SystemProperty})$ is the set of sequences of actions from $\text{sig}(\text{SystemProperty})$ such that:
 1. *day* and *night* alternate, beginning with *day*
 2. after each *day* and before the next *night* there must be at least one *sunlight(false)*
 3. the first instance of *day* must be preceded by a *sunlight(true)*

4. after each *night* there must be at least one *sunlight(true)* before the next day
5. *sunlight(b)* has no further constraints.

The composition of the I/O Automata *Terra* and *Sol* results in the I/O Automata *System*:

- System Signature

- Output Actions

- * *sunlight(b)*, where *b* is a Boolean

- * *day*

- * *night*

- Internal Actions

- * *incrementDegrees*

- System States

- degrees, an integer initially 0

- shining, a, boolean initially true

- sun, a, boolean initially false

- newsun, a, boolean initially false

- System Task Partition

- {*sunlight(b)*}

- {*incrementDegrees*}

- {*day, night*}

- System Transitions

- *sunlight(b)*

- precondition: *b*=shining

- effect:

- lastsun := sun

- sun := *b*

- newsun := true

- *day*

- precondition

- newsun = true

- lastsun = false

- sun = true

- effect:

- newsun = false

- *night*

- precondition

- newsun = true

- lastsun = true

- sun = false

- effect:

- newsun = false

- *incrementDegrees*

- effect:

- degrees = degrees+1

- if degrees modulo 360 < 180

- then shining := true else

- shining := false

Does the *System* I/O Automata satisfy the *SystemProperty* trace property?

external(*SystemProperty*)= {*sunlight(b), day, night*}

external(*System*)= {*sunlight(b), day, night*}

SystemProperty.1 *day* must precede *night* because the state variable *lastsun* is

initially false. A precondition for *night* is that *lastsun* is false and therefore *night* cannot be generated before *day*. Thus, *System* cannot generate a sequence of external actions that violate this specification.

SystemProperty.2 A precondition for *day* is that the state variable *sun* be true. A precondition for *night* is that *sun* be false. The only action that updates *sun* is *sunlight(b)*. Thus, *System* cannot generate a sequence of external actions that violate this specification.

SystemProperty.3 *day* cannot be generated until the state variable *sun* is true. *sun* is initially false and System I/O Automata

Since the only action that updates *sun* is *sunlight(b)*, there must be a *sunlight(true)* before the first *day* action can occur. Thus, *System* cannot generate a sequence of external actions that violate this specification.

SystemProperty.4

This specification is a corollary to *SystemProperty.2*

SystemProperty.5

Places no further restrictions on *System*

Therefore, the traces of all executions of *System* are permitted by *SystemProperty*. That is, $\text{traces}(\text{System}) \subseteq \text{traces}(\text{SystemProperty})$ so *System* does in fact satisfy *SystemProperty*.

Shared Variable Types

Shared Variable Types (SVT) can be used to model shared memory that is utilized for communication of data between multiple independent processes. A Shared Variable Types consists of:

- V a set of values
- $v_0 \in V$ the initial value
- I a set of invocations
- R a set of responses
- f a function $\in (I \times V \rightarrow R \times V)$
- Where the invocations and responses happen together as a single application off.

A collection of SVTs is said to be *compatible* if:

- All sets of invocations are disjoint
 - All sets of responses are disjoint
- The *composition* T of a compatible collection of

Shared Variable Types, T_i is given by:

- V the cartesian product of the value sets of the T_i
- v_0 a vector consisting of each v_{0_i}
- I the union of the sets of invocations from each T_i
- R the union of the sets of responses from each T_i

• f a function acting componentwise on each T_i

Two types of SVT are *read-write* and *read-modify-write*. A read-write SVT is like a register. It holds a value that may be written and may be read. A read-modify-write SVT

allows a user to atomically examine the state of the variable, make decisions based on the state and write a new value to the variable.

2.2 Implementation Tools

A decade ago, a distinction was made between system programming languages and scripting languages [9]. System languages such as C, C++ and Java are characterized as general abstractions of the underlying machine that provide static typing. These

features make them good for developing data structures and algorithms from scratch. Over time, system languages derived from Algol (PL/1, Pascal, C, C++ and Java) have supplanted assembly languages for the development of nearly all large applications. The benefits were numerous; register management, stack management and control structures were just a few. Also, with the higher level of abstraction, computer programming was opened up to a larger number of people. There were some initial objections over potentially slower execution but those were swept away by the tides of increased productivity, increased quality and the doubling of processor capability every 18 months.

Scripting languages, by contrast, are interpreted and not compiled. Thus, static typing is not possible and, in fact, not desirable in a scripting language. As described in [9], scripting languages excel in connecting things together. They are “glue” languages. In order to simplify this capability, scripting languages are dynamically typed. A variable may at one moment hold a user defined object, the next a string. The interpreter manages the type conversions at runtime. Because of this flexibility, scripting languages excel at building applications out of pre-existing components. Scripting languages do not have the runtime performance of carefully crafted system languages but still, processor performance improvement continues unabated and today’s machines are fast enough to run many scripted applications. With their still higher level of abstraction, scripting languages are noted for rapid application development. Python [10] is an object oriented scripting language that is used in a wide variety of application domains. Its proponents [11] cite the following advantages.

- very clear, readable syntax
- strong introspection capabilities
- intuitive object orientation

- natural expression of procedural code
- full modularity, supporting hierarchical packages
- exception-based error handling
- very high level dynamic data types
- extensive standard libraries and third party modules for virtually every task
- extensions and modules easily written in C, C++ (or Java for Jython, or .NET languages for IronPython)
- embeddable within applications as a scripting interface

It's also been argued [12] that Python is a nearly ideal first computer language and suitable for an introductory CS1 and CS2 curriculum.

- Simple syntax and semantics. An introductory class should be about learning problem solving, solution design and programming and not about learning the details of a particular language. It should be possible for students to solve simple problems simply. Python is simple. Hello World in Python is programmed as:

```
print "Hello World"
```

Block structure in Python is maintained by level of indentation. There are no curly braces to delineate blocks. Not only is it impossible to forget the braces but it makes for more easily readable source code.

- High level and flexible. Python supports modern approaches to design such as abstraction and object oriented techniques. The class model is a simplification of the C++ model and supports multiple inheritance. Python comes with high level built in data types, e.g. list (dynamic array), tuples (immutable lists) and dictionaries (hash tables). Python comes with a built in module loading system that is simple and clean and dynamically loads modules as necessary.

- Batteries included. With Python comes an extensive library of modules for practically any kind of application development. There are GUI toolkits, web servers, numerical processing, xml libraries etc. The availability of libraries that can be invoked dynamically with a cursory knowledge of Python allows a student to move beyond "toy" applications rapidly.

SimPy [13] is a discrete event simulation package developed in Python. It is a process oriented approach to simulation, where each activity is modeled by a process. Process oriented simulations produce more modular code and are becoming the most popular simulation paradigm; a discrete-event simulation course has been developed with SimPy [14].

SimPy is written in Python and called by Python programs. All processing that happens within a simulation occurs within `SimPy.Process` objects. Each object implements a *Process Execution Method* or PEM. Each PEM is a Python generator function, that is, it contains one or more *yield expression* statements. Generator functions are wrapped in iterator objects that maintain the function's local state and next statement of execution. They are a form of coroutines, each new call to the function resumes execution with the statement immediately following the yield statement.

SimPy process execution methods make use of generators to manage the concurrency inherent in the process oriented approach. The two types of *expressions* that will be utilized in the PEM yield statements are:

- `yield hold 1.0` which deactivates the invoking `SimPy.Process` for one tick of the simulation clock. SimPy reacts to this by scheduling the next statement in the PEM to execute at the next tick of the simulation clock which occurs when there are no more PEMs still active for the current tick.

- `yield passivate self` which deactivates the invoking `SimPy.Process` indefinitely. SimPy won't schedule the next statement in the PEM until an explicit reactivation for this process is conducted by some other PEM.

Basic usage of SimPy is very straight-forward.

1. Import the package `SimPy.Simulation`.
2. Define classes that derive from the `SimPy Process` class.
 - Each class must implement a process execution method with one or more `yield` statements, typically occurring within an infinite loop.
 - Each class must invoke its base class constructor as the first statement in its own constructor.
3. Call the `SimPy initialize` method.
4. Instantiate objects from the defined classes.
5. Call the `SimPy activate` method for each object.
6. Call the `SimPy simulate` method passing the desired length of the simulation.

2.3 Verification Tools

The verification of an execution requires that the observable behavior of the components in question be compared against behavior allowed by the appropriate specifications. In order to accomplish this task the observable behavior must be analyzed in real time or captured, saved and analyzed after the execution. The latter approach is taken here.

In this work Python decorators are the tools that are used to capture the observable behavior during an execution. A decorator is a Python function and any callable object can be annotated with a decorator. Consider the Python program *decSalutation.py*.

```

# print the name of the object that's passed
# in and then return that same object
def printFName(f):
    print f.__name__
    return f
if __name__=='__main__':
    @printFName
    def printHello(): print "Hello" printHello()

    @printFName
    def printGoodbye(): print "Goodbye"
    printGoodbye()

```

The output of running this program is

```

printHello
Hello
printGoodbye
Goodbye

```

Note the *@printFName* annotation of the two functions *printHello* and *printGoodbye*.

These two functions were decorated with the function *printFName*. When each was invoked in *main* it was actually their decorator that was invoked and whatever object it returns is subsequently invoked. We note that the same functionality could have been provided without decorators but the result is boilerplate code that each function needs to implement.

```

def printHello():
    print printHello.__name__
    print "Hello"

```

Python decorators have multiple benefits [15]:

- decorators help reduce boilerplate code;
- decorators help separation of concerns;
- decorators enhance readability and maintainability;

- decorators are very explicit.

The *decorator* module [15] is an add on module that makes it simpler and safer to define signature preserving decorators. There are two decorators defined in this research based on that decorator module. These decorators are implemented in *trace decoraors.py*. The object *tracer* that they refer to will be explained in the next section.

```

from decorator import decorator
@decorator def traceInvocation(f, *args, **kw):
    try:
        f.__getattribute__("hide")
        # the method has the hide attribute
        # so don't trace it
        return f(*args, **kw)
    except(AttributeError):
        from inspect import getargspec
        (fargs, vararg, vakw, defaults) =\
            getargspec(f)
        s=" "
        s+=str(id(args[0]))
        s+="."+f.func_name+"("
        i=1
        sep=" "
        for x in fargs[1:]:
            if fargs.index(x) < len(fargs)-1:
                sep=","
            else:
                sep=" "
            s += x+"="+str(args[i])+sep
            i+=1
        s+=")"
        tracer.debug(s) return f(*args,**kw)

@decorator
def traceResult(f, *args, **kw):
    def newf(*args, **kw):
        res = f(*args, **kw)
        s=" "
        s+=str(id(args[0]))
        s+="."+f.func_name+": "
        s+=str(res)
        tracer.debug(s)
        return res
    newf.func_name=f.func_name

```

```
return newf(*args, **kw)
```

For each component in the system external actions are decorated with the *@traceInvocation* decorator. Their invocations, including the parameters of the invocation, are traced to a logging system for subsequent examination. For each shared variable type their invocations and responses are captured by decorating the appropriate methods with both the *@traceInvocation* and the *@traceResult* decorators. The Python logging module [16] provides functions and classes that implements a flexible logging system for use by Python applications. This feature is used to record the information obtained by the decorators described previously. Logging is performed by calling method on a *Logger* instance which records *LogRecords* which are dispatched by *Handlers* to their appropriate location. The module *traceDecorators.py* defines the logging functionality implemented in this effort.

```
import logging
from logging import handlers
logging.basicConfig(level=logging.DEBUG)
traceFileHandler= \
    logging.FileHandler('./traces.trace', 'w')
traceFileHandler.setLevel(logging.DEBUG)
traceMemoryHandler=logging.handlers.MemoryHandler \
(1000, logging.DEBUG, traceFileHandler)
tracer=logging.getLogger('traces') tracer.propagate=0
tracer.addHandler(traceMemoryHandler)
traceLines=None
def flushTrace():
    traceMemoryHandler.close()
def getTraceFile():
    return './traces.trace'
def getTraceLines():
    global traceLines
    if traceLines is None:
        trace=open(getTraceFile(), "r")
```

```
traceLines=trace.readlines() trace.close() return
traceLines
```

There are two types of handlers defined for the logging. During runtime LogRecords are recorded in memory by a *MemoryHandler* and then flushed to a file (traces.trace) when the function *flushTrace* is called. The traces are subsequently retrieved as part of the verification process.

The Python *unittest* module [17] is used to drive the verification of captured and logged traces of an execution. The *unittest* module provides a framework for developing and executing test code without impacting implementation code. It enforces separation of the test/verification code from the functional code.

Modules that implements verification code define a subclass of the *unittest.TestCase* class. Each subclass defines one or more test cases by defining *test* case methods. These methods begin with *test* and are invoked by the unittest module framework. Each *test* method returns a boolean value when called by an assertion method in the framework. For instance, *failUnless* will cause the test to pass or fail.

When the verification modules are invoked after an execution by the *unittest* framework each the conditions required by the specification for that layer are asserted by the *test* methods and if any assertion fails then the execution did not satisfy the specification.

2.4 Example

Recall the *System* I/O Automata and the *SystemProperty* trace property that it satisfied. What follows is an implementation in Python using the suite of implementation and verification tools just described. At the module level the necessary tools are imported.

```

from SimPy.Simulation import *
from trace_decorators import *
import unittest

```

Next, also at the module level is the definition of the class that implements the *System*

I/O Automata.

```

class System(Process):
    def __init__(self):
        Process.__init__(self)
        self.degrees=0
        self.shining=True
        self.sun=False
        self.lastsun=False
        self.newsun=False

    def execute(self):
        while True:
            self.sunlight(self.shining)
            if self.newsun and self.sun \
            and not self.lastsun:
                self.day()
            elif self.newsun and self.lastsun \
            and not self.sun:
                self.night()
            yield hold, self, 1.0
            self.incrementDegrees()

    def incrementDegrees(self):
        self.degrees+=1
        if(self.degrees % 360) < 180:
            self.shining=True
        else: self.shining=False

    @traceInvocation
    def sunlight(self, b):
        self.lastsun=self.sun
        self.sun=b
        self.newsun=True

    @traceInvocation
    def day(self):
        self.newsun=False

```

```

@traceInvocation
def night(self):
    self.newsun=False

```

Note how the *states* and *start* are defined simply in the *init* method which is the object's constructor. Since *System* derives from *SimPy.Process* it must call its base constructor first. Also, it must have a process execution method, which in this case is called *execute*. Encompassed in the PEM is the task partition, *tasks*, of locally controlled actions. External actions are decorated with *@traceInvocation* trace decorators so they will show up in the execution traces. This is a demonstration of the synergy created by I/O Automata, Python, SimPy and the decorator and logging facilities available in Python. They simply and clearly allow the composition and of an executable version of the system. The implementation class is a straight-forward translation from the I/O Automata. Importantly, the I/O Automata can be recovered from the Python implementation class just as simply, thereby letting the implementation become its own specification.

It can also be demonstrated that any execution of *System* satisfies *SystemProperty*. This is done by defining a *unittest.TestCase* that verifies that the property was not violated during an execution.

```

SUNLIGHT=0 DAY=1 NIGHT=2
class VerifySystemTraces(unittest.TestCase):

    def setUp(self):
        self.traceLines=getTraceLines()

    """ SystemProperty (SP):
    -Sig(SP) = {sunlight(b), day, night} where b is a
    boolean
    -Traces(SP)
    1. day and night alternate beginning with day
    2. After each day there must be at least one
        sunlight(False) before the next night.

```

3. The first instance of day must be preceded by a
sunlight(True)
4. After each night there must be at least one
sunlight(True) before the next day. """

```
def testSystemProperty(self):
    def lineType(line):
        try:
            line.index("sunlight") return SUNLIGHT
        except(ValueError): pass
        try:
            line.index("day") return DAY
        except(ValueError): pass
        try:
            line.index("night") return NIGHT
        except(ValueError):
            raise Exception("Unknown line type")
    def doVerifySP1(lines):
        expecting=DAY
        for line in lines:
            if expecting==DAY and \
               lineType(line)==NIGHT:
                return False
            if expecting==NIGHT and \
               lineType(line)==DAY:
                return False
            if lineType(line)==DAY:
                expecting=NIGHT
            elif lineType(line)==NIGHT:
                expecting=DAY
            return True
    def doVerifySP2(lines):
        gotDay=False
        gotSunlightFalse=False
        for line in lines:
            if lineType(line)==DAY: gotDay=True
            elif lineType(line)==SUNLIGHT:
                lineparts=line.split("=")
                b=lineparts[1].split("")
                if b[0]=="False":
                    gotSunlightFalse=True
            else:
                if gotDay and gotSunlightFalse:
                    gotDay=False
                    gotSunlightFalse=False
                else:
                    return False
        return True
```

```

def doVerifySP3(lines):
    gotSunlightTrue=False
    for line in lines:
        print line
        if lineType(line)==DAY:
            if gotSunlightTrue: return True
            else: return False
        elif lineType(line)==SUNLIGHT:
            lineparts=line.split("=")
            b=lineparts[1].split(" ")
            if b[0]=="True":
                gotSunlightTrue=True
    return True
def doVerifySP4(lines):
    gotNight=False
    gotSunlightTrue=False
    for line in lines:
        if lineType(line)==NIGHT: gotNight=True
        elif lineType(line)==SUNLIGHT:
            lineparts=line.split("=")
            b=lineparts[1].split(" ")
            if b[0]=="True":
                gotSunlightTrue=True
        else:
            if gotNight and gotSunlightTrue:
                gotNight=False
                gotSunlightTrue=False
            elif gotNight:
                return False
    return True
self.failUnless(doVerifySP1(self.traceLines))
self.failUnless(doVerifySP2(self.traceLines))
self.failUnless(doVerifySP3(self.traceLines))
self.failUnless(doVerifySP4(self.traceLines))
return True

```

VerifySystemTraces defines a single *test* method which gets invoked by the *unittest* framework. However, there are four separate conditions that are checked within that single test case so that all of the properties of *VerifySystemTraces* are demonstrated to hold.

With the implementation and verification code in place it is necessary to define the test harness, the driver program, which will execute the implementation and verification functionality.

```
if __name__ == '__main__':
    system=System()
    initialize()
    activate(system, system.execute())
    simulate(until=720)
    flushTrace()
    suite = unittest.TestSuite()
    suite.addTest(unittest.makeSuite(VerifySystemTraces
    ))
    unittest.TextTestRunner(verbosity=2).run(suite)
```

For this example:

- External behavior has been specified with trace properties. The trace properties defined a signature and a permissible set of sequences of external actions.
- From the trace properties, a compatible implementation specification, represented as an I/O Automata, was defined. The automaton's signature was compatible with the trace property. The automaton was developed by defining the states and actions such that the trace property would be satisfied by executions of the implementation.
- From the I/O Automata implementation specification, a very simple translation to a SimPy.Process was performed which made the specification executable. It implemented the specification.
- Executing the SimPy.Process enabled the traces of the execution to be captured, stored and subsequently verified by the verification tools; decorator, logging and unittest modules.

This example has demonstrated a *linked* set of tools. A tool *set*, in fact, sufficient to progress through the specification, implementation and verification phases without ever leaving the tool set. Furthermore, this tool set is usable by anyone capable of understanding some basic set theory and learning the Python scripting language. The tool set is *end to end* and *readily accessible*. The rest of this thesis will show that this toolset and approach can be applied to specifying, implementing and verifying realistic layered network simulations.

CHAPTER III

MODEL

In order to demonstrate that the method defined and illustrated in the previous chapter is applicable to the domain of layered network protocols, a realistic model must be employed. The model must capture enough elements of a real layered network protocol so that its specification, implementation and verification establish the viability of the methodology [8].

The 802.11 [18] protocol for wireless communication between stations, supplies the necessary components for establishing a target model that satisfies requirements. 802.11 is a layered protocol, with the two main layers being the Medium Access Control (MAC) and the Physical Service (PHY) layers. An architectural goal [19] of 802.11 was to define a single MAC layer that could make use of multiple PHY layers. Accordingly, within the PHY layer there are additional layers defined which has enabled the 802.11 protocol to evolve through various versions, e.g. 802.11, 802.11b, 802.11a, 802.11g, and 802.11n with changes being isolated to the PHY. Leveraging a single, stable MAC layer, the advancement of the 802.11 protocol has happened at the physical layer and it is this layer that serves as the basis for the model in this paper. The requirements of being realistic and compositional are satisfied by such a basis.

3.1 The Wireless Medium Layer

The lowest layer in the model is called the wireless medium. This layer of the model defines how a signal is sent from one station to another. Within the model it represents the physical world. As such, it provides the model with a notion of space (a place for the signal to travel) and time (when the signal should travel). Additionally, physical modifications to the signal that occur in transit should be modeled at this layer.

In the wireless medium layer, space is modeled as a read-write shared variable type through which stations transmit and receive bits of data. The shared variable is independently written to and read from. This is analogous to a station antenna transmitting and receiving electromagnetic waves at a given frequency.

Time is modeled in the wireless medium layer by design. A SimPy process at this layer will awake every simulation clock tick and send a *tick* upwards through the model stack. This is the only layer that functions based on a time clock; all other processes are reactive in nature and respond to the *tick* by examining local state and performing their functions accordingly.

The ability to handle modifications to a transmitted signal are demonstrated at this layer by detecting signal collisions. That is, when more than one station is transmitting on the wireless medium the medium detects the collision and carries a garbled signal. The mechanism provided for this functionality, a read-modify-write shared variable type, however makes it possible to expand handling to other physical situations as well since the state of the medium is made available to users as part of the invocation.

3.2 The Physical Medium Dependency Layer

The purpose of the Physical Medium Dependency (PMD) layer is the transmission and reception, between independent stations, of binary data across a wireless medium. The PMD layer is responsible for turning data into signals (transmission) and signals into data (reception). In the 802.11 domain this layer has multiple specifications and implementations, e.g. Direct Sequence Spread Spectrum (DSSS), Frequency Hopping Spread Spectrum (FHSS) and Orthogonal Frequency Division Multiplexing (OFDM).

The model PMD layer will transmit and receive signals from a set of signals that the wireless medium recognizes and handles. It will do this based on the commands that it receives from higher layers in the model, making use of the features that the wireless medium model provides. In the realm of wireless communication, transmission and reception are independent modes of operation since the signal strength of a transmitting antenna overwhelms any signal that may be coming in from another station. Thus, transmission and reception are handled by different, independent processes in the PMD layer.

3.3 The Wireless Channel Layer

The wireless channel layer is unique to this model and is not part of the 802.11 set of protocols. Its purpose is to provide a composition of the PMD layer receive and transmit processes so that higher layers in the model can invoke any request on a single object. This layer, in effect, represents an asynchronous shared memory model [7]. Although called a wireless channel, since our model is based on 802.11, it is in fact a channel of communication for bit oriented data through a single shared variable. So, beyond its support for simple usage of the PMD by higher layers, it provides a

mechanism for further work on parallel communication systems since, as I/O Automata, N multiple channels may be composed each of which communicates through one of N unique shared variables.

3.4 The Physical Layer Convergence Procedure

With the goal of a single MAC layer driving multiple PHY layers, 802.11 placed a burden on the PHY layer to provide a single interface for use by the MAC. That interface specifies the requirements that all physical layers must satisfy regardless of the differences in wireless medium and PMD layers. The satisfaction of this interface is the responsibility of the Physical Layer Convergence Procedure (PLCP) layer. These requirements include:

- The physical service must report the status of the medium. Once a station starts transmitting, it is not possible for that station to detect a collision, since at that point in space and time its own signal overwhelms the garbled signal that results farther from its own location. Once competing MAC layers start transmitting data, they will continue, oblivious to the collision. Therefore, the MAC protocol is based on collision avoidance rather than collision detection and the physical service must support this by continually updating the MAC with the status of the medium.

- The MAC layer sends the data that it wishes transmitted an octet at a time to the PHY. It is the responsibility of the PLCP layer, which implements the physical service, to transform these octets into bit streams and transmit them a bit at a time in the proper order. The MAC layer also receives data an octet at a time from the PHY. It is the responsibility of the PLCP layer to collect received bits and collate them into octets for presentation to the MAC layer.

Beyond the requirements specified for usage by the MAC layer, the PLCP is also responsible for sending and receiving the bits in *frames* which coordinates the communication.

CHAPTER IV

SPECIFICATION

The first step in developing a system is to identify the requirements that the system must satisfy. A specification of the system behavior must be created. Using the tools selected, this section defines the specification for the allowable behavior at each layer identified in the system model.

4.1 Wireless Medium Specification

The correct behavior of the wireless medium layer is specified by three trace properties, *VotaProperty*, *NumTxProperty* and *WirelessMediumProperty* each of which coincides with a component that is part of the implementation.

SVT_Vota is a wireless medium component that maintains the value on the air. It is a read-write shared variable type. When a station wishes to transmit a value, it ultimately invokes SVT_Vota.write(*v*) which places the value *v* into the shared variable. When a station wishes to receive a value, it ultimately invokes SVT_Vota.read() which retrieves the current value, *v*, from the shared variable. A trace property, *VotaProperty*, can be defined for SVT_Vota to assure that only valid values are stored and that values are maintained and not lost.

- $\text{Sig}(\text{VotaProperty}) = \{\text{invocation}, \text{response}\}$ where:
 - $\text{invocation} \in \{\text{read}, \text{write}(v)\}$
 - $\text{response} \in \{(\text{votaRead}, v)(\text{votaAck}, v)\}$
- $\text{Traces}(\text{VotaProperty})$
 - The set of all sequences of invocations and responses on SVT_Vota such that
 - $v \in \{\text{ZERO}, \text{ONE}, \text{NO_DATA}, \text{INDETERMINATE}\}$
 - $\forall (\text{votaRead}, v)$ responses, v is equal to the value of v in the previous $\text{write}(v)$ invocation.

Thus, the correct behavior of SVT_Vota can be described as maintaining valid values that are written to it and supplying those values when read.

SVT_NumTx is a wireless medium component that is used to keep track of how many stations are transmitting at any time. SVT_NumTx is a read-modify-write shared variable type. That is, users who call its invocations first receive a copy of SVT_NumTx's state and can then make a decision on what function SVT_NumTx should execute creating the appropriate response, all of which happens atomically. In the target system, the only function passed to SVT_NumTx to execute is an increment function. The function, when executed, will increment, in SVT_NumTx, the number of stations that are transmitting. This allows for the detection of collisions when more than one station is transmitting. SVT_NumTx is a read-modify-write shared variable type in order to demonstrate a mechanism for adjusting physical properties such as signal strength, or location based on the state of the wireless medium. Future work may exploit this feature.

A trace property, *NumTxProperty*, can be defined for SVT_NumTx to assure that it honors the atomic nature of a read-modify-write shared variable type and that it maintains the correct number of transmitters based on invocation history.

- $\text{Sig}(\text{NumTxProperty}) = \{\text{invocation}, \text{response}\}$ where:
 - $\text{invocation} \in \{g(\text{func} = \text{increment})\}$
 - $\text{response} \in \{(\text{numT } x, v)\}$
- $\text{Traces}(\text{NumTxProperty})$
 - The set of all sequences of invocations and responses on SVT_NumTx such that invocations and responses alternate, beginning with an invocation.
 - The set of all sequences of invocations and responses on SVT_NumTx such that: $\forall (\text{numT } x, v) \text{ responses, } v \text{ is equal to the number of previous increment invocations.}$

The final wireless medium component specification is SVT_WirelessMedium. It composes SVT_Vota and SVT_NumTx and exposes their invocations and responses to the direct users of the wireless medium. There is a direct mapping of *SVT_WirelessMedium* invocations and responses onto SVT_Vota and SVT_NumTx and the same trace properties hold for those components.

As the target system model's component of physical reality SVT_WirelessMedium adds the notion of time to the physical layer. It provides a simulation time tick at the start of every simulation round when SimPy increments its internal clock. It models "real" time for the system, every simulation time tick gets sent by the wireless medium to all PMD components. This allows for the management of the execution order of PMD layer transmitters and receivers. The target system has one bit transmission per tick of the simulation clock. Thus, PMD layer transmitters need to execute first so that collisions can be detected before the receivers read the data. This enforces clock synchronization between stations as a simplifying assumption. Future

work can explore the behavior of relative clock drift by adjusting the value of `const.SIMTICKS PER BIT`.

This simplification allows the rest of the system to be designed with an explicit mechanism for managing concurrency. Receivers are activated bottom up. That way receiving processes at lower layers read data and have it ready for when the higher layer receivers need it. Conversely, transmitters are activated top down. That way, when a low layer transmitter becomes active, the higher layer transmitter process has already supplied the data to be transmitted.

A trace property *WirelessMediumProperty* unique to *WirelessMedium* assures that for each simtick that the registered transmitters are activated before the registered receivers and that multiple simultaneous transmissions are detected and reflected in the state of the medium.

- $\text{Sig}(\text{WirelessMediumProperty}) = \{\text{invocation, response}\}$ where:
 - $\text{invocation} \in \{\text{simtick}(\text{simtime}), g(\text{func} = \text{increment}), \text{writeVota}(v)\}$
 - $\text{response} \in \{\text{writeVota} : (\text{votaAck}, v), S\}$ where: $S = [t1.\text{simtick} \dots, tn.\text{simtick}, r1.\text{simtick}, \dots, rn.\text{simtick}]$
- $\text{Traces}(\text{WirelessMediumProperty})$
 - The set of all sequences of alternating invocations and responses such that between invocation $\text{simtick}(\text{simtime})$ there is a corresponding response S . This assures that transmitting and receiving processes were activated in the proper order.
 - The set of all sequences such that after each $\text{simtick}(\text{simtime})$ when there has been more than one $g(\text{func} = \text{increment})$ then for each $\text{writeVota} : (\text{'votaAck'}, v)$ before the next $\text{simtick}(\text{simtime})$ that $v = \text{INDETERMINATE}$. This assures that

whenever more than a single transmitter process transmits within a simtick that the collision is detected and modeled.

Thus, *WirelessMediumProperty* composes *VotaProperty* and *NumTxProperty* and adds the abilities to assure correct activation of dependent processes and detecting conflict arising from multiple simultaneous transmissions. These three trace properties, in conjunction, specify the correct execution of the wireless medium layer of the target model.

4.2 Physical Medium Dependency Specification

The Physical Medium Dependency (PMD) layer consists of receiver and transmitter processes that utilize the wireless medium in order to receive and transmit data bits. These processes interact with the wireless medium according to the rules inherent in the wireless medium design. Therefore, for instance, processes at the PMD layer must receive *simticks* from the wireless medium. The PMD layer must also define the interface for use by the higher layers and specifying the correct usage of that interface.

The correct behavior of the physical medium dependency layer is specified by two trace properties, *TransmitterProperty* and *ReceiverProperty* each of which coincides with a SimPy process that is part of the implementation.

TransmitterProperty defines the correct behavior for the transmitter process of the PMD layer. It is responsible for providing a transmit interface for composition by higher layers and composing the transmit actions of *WirelessMediumProperty*. It assures that usage of the transmit function in the PMD in conjunction with usage by that same transmit function of the wireless medium results in each execution of `pmd_impl.py` in the correct data being transmitted at the correct time across the wireless medium.

- $\text{Sig}(\text{TransmitterProperty}) = S$ where: $S = \{\text{simtick}(\text{simtime}), \text{pmd_data_request}(\text{bit}), \text{pmd_tx_start}, \text{pmd_tx_end}, \text{writeVota}(\text{value}), \text{datatick}\}$ •
- $\text{Traces}(\text{TransmitterProperty})$ is the property such that $\forall$ sequences of actions, a , where $a \in S$ the following hold: $\forall$ datatick the total number of preceding simticks modulo number of simticks per bit = 0.

Thus, *TransmitterProperty* assures that the correct data bit is transmitted when it was supposed to be transmitted.

ReceiverProperty defines the correct behavior for the receiver process of the PMD layer. It is responsible for monitoring the status of the wireless medium and reporting that status to the higher layers. The status that it reports includes two items; whether there is someone transmitting (carrier sense) and whether a bit (const.ZERO or const.ONE) has been demodulated. It also responsible for composing the *WirelessMediumProperty* receive related actions.

- $\text{Sig}(\text{ReceiverProperty}) = S$ where: $S = \{\text{simtick}(\text{simtime}), \text{pmd_data_indicate}(\text{bit}), \text{pmd_cs_indicate}(\text{cs}), \text{readVota}\}$
- $\text{Traces}(\text{ReceiverProperty})$ is the property such that $\forall$ sequences of actions, a , where $a \in S$ the following hold:
 - $\forall$ readVota there is a unique preceding simtick .
 - $\forall$ $\text{pmd_cs_indicate}(\text{cs})$ there is a unique preceding readVota .
 - $\forall$ $\text{pmd_data_indicate}(\text{bit})$ $\text{bit} = \text{const.ZERO}$ or $\text{bit} = \text{const.ONE}$.

Thus, *ReceiverProperty* assures that the wireless medium is monitored continuously while the receiver is active, that it continuously reports the status of medium and that only a valid demodulated bit of data gets reported.

4.3 Wireless Channel Specification

The Wireless Channel (WC) layer composes the PMD Transmit and PMD Receive processes from the PMD into a single component. It provides the functionality of the asynchronous shared memory model described in [7]. It adds no additional processing, it is a wrapper and a composition. A user of the wireless channel layer needs only create an instance of a `WirelessChannel` class and they will have the ability to transmit and receive bits of data across the wireless medium from a single component. Generally a station will have one `WirelessChannel` but it is possible in future work to develop multiple, simultaneous wireless channels, each with their own wireless medium. This would allow modeling of systems that transmit multiple bits in parallel.

The correct behavior of the wireless channel layer is specified by *WirelessChannelProperty*. It is responsible for composing the *ReceiverProperty* and the *TransmitterProperty* from the PMD layer. It exposes equivalent actions to higher layers.

- $\text{Sig}(\text{WirelessChannelProperty}) = S$ where: $S = \{\text{datatick}(), \text{pmd_data_indicate}(\text{bit}), \text{pmd_cs_indicate}(\text{cs}), \text{pmd_data_request}(\text{bit}), \text{pmd_tx_start}(), \text{pmd_tx_end}(), \text{wc_tx_data_request}(\text{bit}), \text{wc_tx_start}(), \text{wc_tx_end}(), \text{wc_datatick}(), \text{wc_data_indicate}(\text{bit}), \text{wc_cs_indicate}(\text{cs})\}$
- $\text{Traces}(\text{WirelessChannelProperty})$ is the property such that $\forall$ sequences of actions, a , where $a \in S$ the following hold:
 - $\text{datatick}()$ and $\text{wc_datatick}()$ alternate, beginning with $\text{datatick}()$
 - $\text{pmd_data_indicate}(\text{bit})$ and $\text{wc_data_indicate}(\text{bit})$ alternate, beginning with $\text{pmd_data_indicate}(\text{bit})$ and for each $\text{pmd_data_indicate}(\text{bit})$ the following $\text{wc_data_indicate}(\text{bit})$ will have *bit* of the same value.

– pmd_cs_indicate(cs) and wc_cs_indicate(cs) alternate, beginning with pmd_cs_indicate(cs) and for each pmd_cs_indicate(cs) the following wc_cs_indicate(cs) will have *cs* of the same value.

– wc_tx_data_request(bit) and pmd_data_request(bit) alternate, beginning with wc_tx_data_request(bit) and for each wc_tx_data_request(bit) the following pmd_data_request(bit) will have *bit* of the same value.

– wc_tx_start() and pmd_tx_start() alternate, beginning with wc_tx_start().

– wc_tx_end() and pmd_tx_end() alternate, beginning with wc_tx_end().

These properties assure that when the *WirelessChannel* is accessed, either from the PMD or from a higher layer, that it translates the request into the appropriate one for the other layer before an intervening request can interfere. *WirelessChannel* is a bridge from one layer of active processing, PMD, to another, the Physical Layer Convergence Procedure (PLCP).

4.4 Physical Layer Convergence Procedure Specification

The Physical Layer Convergence Procedure (PLCP) is responsible for implementing the Physical Data Service (PHY). It is responsible for both transmitting and receiving data. When it transmits it cannot receive. Since transmission and reception cannot occur at the same time, there are two modes of operation for the PLCP layer. When it is commanded from the higher layers to commence with a transmission it switches from receive mode to transmit mode. When transmitting, it is responsible for converting bytes to bits and packaging the bits into a frame that can be received by other stations. When done transmitting, the PLCP layer switches to receive mode. When in receive mode, it is responsible for indicating to the higher layers whether or not the channel is clear and for converting received bits into bytes and conveying them to the

higher layer. There are two trace properties that specify correct behavior of the physical layer convergence procedure, *PlcpTransmitProperty* and *PlcpReceiveProperty*.

PlcpTransmitProperty defines the correct behavior for the PLCP when it is in transmit mode. It provides an interface to the higher layers to start and end transmissions and to transmit an octet of data.

- $\text{Sig}(\text{PlcpTransmitProperty}) = S$ where: $S = \{\text{wc_tx_data_request}(\text{bit}), \text{wc_tx_start}, \text{wc_tx_end}, \text{tx_data_request}(\text{octet}), \text{tx_start_request}(\text{numOctets}), \text{tx_end_request}, \text{tx_start_confirm}, \text{tx_end_confirm}, \text{tx_data_confirm}\}$
- $\text{Traces}(\text{PlcpTransmitProperty})$ is the property such that $\forall$ sequences of actions, a , where $a \in S$ the following hold:
 - Each tx_start_request will be followed by one wc_tx_start which will be followed by one tx_start_confirm which will be followed by a tx_end_confirm .
 - Each tx_end_request will be followed by one wc_tx_end .
 - After a tx_end_request there will be no $\text{wc_tx_data_request}$ until after a tx_start_request . Each tx_data_request will be followed by eight $\text{wc_tx_data_requests}$ unless a tx_end_request is received.
 - Each tx_data_request will be followed by a tx_data_confirm unless a tx_end_request is received.

PlcpReceiveProperty defines the correct behavior for the PLCP when it is in receive mode. It provides an interface to the WirelessChannel to receive bits that it assembles into octets which are sent to the higher layers. It is also responsible for monitoring the state of wireless, `const.IDLE` or `const.BUSY` and delivering that status to the higher layers.

- $\text{Sig}(\text{PlcpReceiveProperty}) = S$ where: $S = \{\text{wc_data_indicate}(\text{bit}), \text{wc_cs_indicate}(\text{cs}), \text{cca_indication}(\text{cca}), \text{rx_start_indication}(\text{datalength}), \text{rx_data_indication}(\text{data}), \text{rx_end_indication}(\text{rxerror})\}$
- $\text{Traces}(\text{PlcpReceiveProperty})$ is the property such that $\forall$ sequences of actions, a , where $a \in S$ the following hold:
 - $\text{rx_start_indication}$ and rx_end_indication alternate, beginning with $\text{rx_start_indication}$.
 - Between each $\text{rx_start_indication}$ and rx_end_indication , after every eighth wc_data_indicate there is an $\text{rx_data_indication}$ unless there was a carrier loss.
 - For each $\text{cca_indication}(\text{cca})$ if $\text{cca} = \text{const.CCA IDLE}$ then the previous $\text{wc_cs_indicate}(\text{cs})$ had $\text{cs} = \text{False}$. If $\text{cca} = \text{const.CCA BUSY}$ then the previous $\text{wc_cs_indicate}(\text{cs})$ had $\text{cs} = \text{True}$.

CHAPTER V

IMPLEMENTATION

Once a system has been specified it may be implemented and verified. This section describes the approach utilized as well as the actual results of the implementation effort. The approach defines conventions that provide for a consistent modular approach to implementing and verifying the functionality required at each layer.

Implementation Module

The runtime functionality of each layer is provided in an implementation module. By convention the implementation modules are named *layername_impl.py*. The classes in the implementation module contain the methods that are invoked during a simulation. Generally these methods fall into three categories:

1. Process Execution Methods: these are the methods that are invoked by SimPy.
2. Actions Methods: these are methods that comprise the signature of the I/O Automata that the class is implementing. If the methods are a part of the external signature then their invocation is captured in the trace of the execution.
3. Helper Methods: these are defined to enhance the code organization. Objects from the implementation module are instantiated by the test module that is driving the simulation.

Verification Module

Each layer has a verification module that examines the traces for that layer in a simulation. By convention, the verification modules are named *layername_verify.py*. The

verification module verifies that the traces satisfy the trace properties that have been specified for that layer.

The classes in the verification modules are subclasses of Python supplied `unittest.TestCase`. Leveraging this class keeps the implementation of the trace verification classes clean of code that would be needed to implement a test harness. Trace property verification methods are coded to return a boolean value, *true* if the traces satisfy the trace property and *false* otherwise. These methods are passed, one at a time into the `unittest.TestCase` test harness which will run them in turn and report the results.

Test Module

Each layer has a test module that is used for unit testing the implementation and driving simulations of that particular layer. By convention, the test modules are named *layername_test.py*.

Each test module does essentially the same thing.

1. Provide a starting point for program execution.
2. Import the classes needed to execute the simulation and capture traces.
3. Define clients of this layer's implementation. These clients provide rudimentary functionality needed to drive the implementation. As a side benefit, these clients begin to explore the functionality needed at the next higher layer.
4. Set up the parameters for the simulation(s).
5. Run the simulation(s).
6. Execute the verify module for the same layer.

5.1 Wireless Medium Implementation

The wireless medium layer sits at the bottom of the system stack. It represents the physical world in the layered system specification and implementation. As such it is the

first layer to be specified and implemented. The functionality defined in the wireless medium layer is the foundation for transmitting and data bits from one entity to another within the system.

Wireless Medium Implementation

The implementation of the wireless medium layer must satisfy trace properties *VotaProperty*, *NumTxProperty* and *WirelessMediumProperty*.

In order to satisfy *VotaProperty*, SVT_Vota must have invocations named read and write(v) with corresponding responses (votaRead, v) and (votaAck, v) and the sequence of invocations and responses must be such that only allowable values of v are present in the trace. As seen below, SVT_Vota is derived from SVT_ReadWrite which exposes the required external methods and ensures that invocations and responses are traced by use of Python decorators @traceInvocation and @traceResult. Python tuples are used as the responses allowing the implementation to model the defined structure of a shared variable type response.

```
class SVT_ReadWrite:
    def __init__(self, value, rrsp="read", wrsp="ack"):
        self.value=value
        self.rrsp=rrsp
        self.wrsp=wrsp

    @traceInvocation
    @traceResult
    def read(self):
        return (self.rrsp, self.value)

    @traceInvocation
    @traceResult
    def write(self, v):
        if not self.valueAllowed(v):
            raise Exception("value "+str(v)+" not
                            allowed")
        self.value=v return (self.wrsp, self.value)
```

```

        def valueAllowed(self, v):
            return True

class SVT_Vota(SVT_ReadWrite):

    NONE=const.NO_DATA
    INDETERMINATE=const.INDETERMINATE
    allowable=[0, 1, NONE, INDETERMINATE]

    def __init__(self, value):

        if not self.valueAllowed(value):
            raise Exception("illegal initial value")

SVT_ReadWrite.__init__(self, value, "votaRead",
    "votaAck")

    def valueAllowed(self, v):
        return v in SVT_Vota.allowable

```

In order to satisfy *NumTxProperty*, SVT_NumTx must have an invocation named *g(f)* where *f* = increment. That is, it must expose a method named *g* that takes a function as an argument. Correct usage of SVT_NumTx will pass in a function named *increment* to *g*. Since *g* is decorated by *@traceInvocation* and *@traceResult* its invocation/response pairs will be traced. When invoked, *g(func)* simply invokes *func* on its own instance of SVT_NumTx. This permits arbitrary behavior and changes to this instance to be supplied by a client based on the state of this element of the wireless medium. The functionality permitted by *NumTxProperty* is to increment the number of transmitters and return the number of other transmitters that are currently transmitting.

```

# read-modify-write shared variable type
class SVT_NumTx:

    def __init__(self, value, rsp="numTx"):
        self.value=value
        self.rsp=rsp

    @traceInvocation

```

```

@traceResult
def g(self, func):
    (response, v)=func(self)
    return (response, v)

# support function
def getValue(self): return self.value

```

In order to satisfy *WirelessMediumProperty*, SVT_WirelessMedium must have a an invocation of simtick(simtime) and the response must be an ordered list of simticks invoked on transmitter objects followed by an ordered list of simticks invoked on receiver objects. simtick(simtime) is the simulation time tick. It models real time. Hence, this class models space by composing SVT_Vota and it models time by sending simulation ticks throughout the rest of the system. It provides the physical model for the rest of the system.

One receiver of this output action is the PMD layer. The 802.11 specification says that the PMD layer provides the data clock for the system. That data clock is implemented based on the inputs that this physical clock provides. The PMD data clock ticks as a function of simtick and const.SIMTICKS PER BIT.

```

@traceInvocation
def simtick(self, simtime):

    self.vota.value=const.NO_DATA
    self.numtx.value=0

    # Transmitters first and in order.
    # Then receivers in order.

    # SimPy will execute processes in the order that they
    # reactivate themselves, which by design is the order
    # that they receive their simtick.
    for transmitter in self.transmitters:
        transmitter.simtick()
    for receiver in self.receivers:
        receiver.simtick()

```

Throughout the entire system the wireless medium has the only SimPy Process Execution Method that *yield holds* for a time period. All other PEMs *yield passivate, self*, i.e., wait indefinitely and become reactivated based on *simtick*. These other PEMs then examine their internal state and behave accordingly.

```
def simTicker(self):
    yield hold,
    self, 1.0
    while(True):
        self.simtick(now())
        yield hold, self, 1.0
```

Wireless Medium Verification

The module `wm verify.py` asserts that the trace properties *VotaProperty*, *NumTxProperty* and *WirelessMediumProperty* are true on the trace of an execution of the implementation class in module `wm impl.py`. It does this by defining class *VerifyWMTraces* as a subclass of `unittest.TestCase`. As an instance of `unittest.TestCase`, *VerifyWMTraces* can be installed in a `unittest.TestSuite` and run by a `unittest.TextTestRunner`. It defines three *test* methods that get invoked.

- `testVotaTraces` verifies that trace property *VotaProperty* was not violated during the execution of `wm impl.py`. Defined within `testVotaTraces` are a number of helper functions. Verification begins by collecting all SVT_Vota trace lines, as well as the SVT_WirelessMedium.simtick lines that partition them, into the list variable `votaLines`.

After the appropriate trace entries are retrieved `VerifyWMTraces.testVotaTraces` runs the verification of them by invoking `unittest.TestCase.failUnless(allvValid(votaLines))`. *VotaProperty* requires that all values of *v* be valid in both the invocations and the responses. The function `allvValid` collects all of the invocations and responses for each

simtick and verifies that each collection does not violate the trace property by invoking `allValidForSimtick` on each.

- `testNumTxTraces` verifies that trace property *NumTxProperty* was not violated during the execution of `wm impl.py`. There are two conditions that the traces of an execution must satisfy in a correct usage of *SVT_NumTx*.
 1. The invocations and the responses must alternate, beginning with an invocation. `testNumTxTraces` verifies this by first collecting all of the *SVT_NumTx* trace lines and then using `unittest.TestCase.failUnless` to run `invRspAlt`.
 2. The value of *v* in each response must be equal to the number of increments since the most recent simtick. `testNumTxTraces` verifies this by first collecting together all of the *SVT_WirelessMedium.simtick* and all of the *SVT_NumTx* trace lines and then using `unittest.TestCase.failUnless` to run `vAlwaysCorrect`.
- `testWMTraces` verifies that trace property *WirelessMediumProperty* was not violated during the execution of `wm impl.py`. There are two conditions that must be verified.
 1. Dependent process must be activated in proper order. The function `usersActivatedInCorrectOrder` is used to verify this on the traces of the wireless medium executions. `usersActivatedInCorrectOrder` is invoked as an argument to the `unittest.TestCase.failUnless`. This assures that users were indeed activated in the required order.
 2. The second requirement of *WirelessMediumProperty* is that whenever there are multiple transmitters during a single simtick that the wireless medium indicates that the value on the air is indeterminate. The verification of this property is

accomplished with the function `vValidWhenMultipleTransmitters` which gets run by `unittest.TestCase.failUnless`. Wireless Medium Test

The test module `wm_test.py` drives the simulation and verification of the wireless medium layer. It defines the necessary components to utilize the wireless medium, establishes the parameters for the simulation, executes the simulation and then creates and runs a `TestSuite` based on `VerifyWMTraces` which is defined in the `wm_verify.py` module.

`SVT_WirelessMedium` is utilized by transmitters and receivers. Within `wm_test.py` are defined `SimtickTransmitter` which is a `TransmitterType` and `SimtickReceiver` which is a `ReceiverType`. Recall that ordering of `simtick` invocations is meaningful and base classes `TransmitterType` and `ReceiverType`, defined in `base_types.py`, provide `wm_impl.py` with the type information necessary to properly order those invocations. Also, `TransmitterType` and `ReceiverType` are derived from `SimPy.Process` so they provide the necessary interface into `SimPy` for independent execution.

As a `SimPy.Process` the implementation of `SimtickReceiver` must include a PEM (Process Execution Method). `SimtickReceiver`'s PEM is named `execute`:

```
def execute(self):  
    self.wm.registerUserForSimticks(self, True)  
    while True:  
        yield passivate, self  
        (response, value)=self.wm.readVota()
```

It simply registers itself with the wireless medium to receive `simticks` and then enters a control loop where it puts itself to sleep. It gets awoken when it receives a `simtick`:


```

@traceInvocation
def simtick(self):
    reactivate(self)

```

After waking it reads the value on the air and then passivates itself again until the next simtick. As a SimPy.Process the implementation of SimtickTransmitter must also include a PEM. SimtickTransmitter's PEM is also named execute:

```

def execute(self):
    self.wm.registerUserForSimticks(self, True)
    while True: yield passivate, self

    if self.data%self.timeBetweenBits==0:
        self.wm.g(self.increment)
        (response, value)= \
            self.wm.writeVota(self.data%2)
        self.data+=1

```

It also registers itself with the wireless medium to receive simticks and then enters a control loop where it puts itself to sleep. It gets awoken when it receives a simtick:

```

@traceInvocation
def simtick(self):
    reactivate(self)

```

After becoming active the PEM determines whether it is supposed to transmit a bit this simtick. If so then it passes its increment function to SVT_WirelessMedium.g so that the number of transmitters can be updated. It then writes a value to the air by invoking SVT WirelessMedium.writeVota before returning to the top of the control loop and putting itself to sleep. loop

Beyond defining the components which make use of the wireless medium, the test module at this layer also defines the parameters of the simulation. Each simulation has a random number of transmitters and receivers (in the range 1-5), a random simulation time and random time between bit transmissions for each transmitter component.

```

from constants import *
from random import
Random rand=Random()
maxNumTransmitters=5
maxNumReceivers=5
numTransmitters=rand.randrange(1, maxNumTransmitters,
    1)
numReceivers=rand.randrange(1, maxNumReceivers, 1)
minSimulationLength=const.SIMTICKS_PER_BIT*20
maxSimulationLength=const.SIMTICKS_PER_BIT*5000
simulationLength=rand.randrange \
    (minSimulationLength, maxSimulationLength, \
    const.SIMTICKS_PER_BIT)
maxTimeBetweenTransmit=numTransmitters* \
    const.SIMTICKS_PER_BIT*5

```

With the receivers and transmitters defined and the simulation paramters established, `wm test.py` is ready to run a simulation of the wireless medium layer.

```

# get access to the wireless medium
from wm_impl import wm

# get SimPy ready
initialize()

# get the wireless medium simTicker activated
activate(wm, wm.simTicker(), prior=True)

# create and activate the transmitters and receivers
for i in range(numTransmitters):
    t=SimtickTransmitter(wm)
    activate(t, t.execute())
for i in range(numReceivers):
    r=SimtickReceiver(wm)
    activate(r, r.execute())

# run the simulation
simulate(until=simulationLength)

# save the trace of the execution for verification
flushTrace()

```

Lastly, with the simulation complete the execution traces can be verified by using `wm_verify.py` and Python's `unittest` module.

```

from wm_verify import *
suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(VerifyWMTraces))
unittest.TextTestRunner(verbosity=2).run(suite)

```

A typical output from a wireless medium simulation execution and verification looks like:

```

C:\ms\thesis\gold\wm>python wm_test.py
numTransmitters=2 numReceivers=1 simulationLength=4466
SimtickTransmitter.timeBetweenBits=5
SimtickTransmitter.timeBetweenBits=9 testNumTxTraces
(wm_verify.VerifyWMTraces) ... ok testVotaTraces
(wm_verify.VerifyWMTraces) ... ok testWMTraces
(wm_verify.VerifyWMTraces) ... ok
Ran 3 tests in 4.641s
OK

```

5.2 Physical Medium Dependency Implementation

The Physical Medium Dependency (PMD) layer sits atop the wireless medium layer in the system stack. It is responsible for transmitting and receiving bits of data over the wireless medium. It must work in accord with the design features of the wireless medium while providing the necessary interface for correct usage by the next higher layer in the stack. The two functions provided by the PMD, transmitting and receiving, are independent of each other. They are therefore implemented as separate SimPy processes.

Physical Medium Dependency Implementation

The PMD layer implementation module is `pmd_impl.py`. This module must satisfy the the trace properties defined for the physical medium dependency layer. These properties are *TransmitterProperty* and *ReceiverProperty*. `pmd_impl.py` accomplishes this by defining two SimPy processes, `PMD_Transmit` and `PMD_Receive`.

In order to satisfy *TransmitterProperty* the SimPy process `PMD_Transmit` must implement and expose an external interface consisting of methods named in the signature

of *TransmitterProperty* and the traces of its possible executions must be permitted by *traces(TransmitterProperty)*. The signature, *S*, of *TransmitterProperty* is given by: *S* = {*simtick(simtime)*, *pmd_data_request(bit)*, *pmd_tx_start*, *pmd_tx_end*, *writeVota(value)*, *datatick*}. As seen below *PMD_Transmit* has a method for each of these actions whose invocations are traced with an *@traceInvocation* decorator.

```

@traceInvocation
def simtick(self):
    self.numsimticks+=1
    if self.numsimticks%const.SIMTICKS_PER_BIT==0:
        self.datatick()
        reactivate(self)

@traceInvocation
def pmd_data_request(self, bit):
    self.databit=bit

@traceInvocation
def pmd_tx_start(self):
    self.txactive=True

@traceInvocation
def pmd_tx_end(self):
    self.databit=const.NO_DATA
    self.txactive=False

@traceInvocation
def writeVota(self, value):
    self.wm.writeVota(value)

@traceInvocation
def datatick(self):
    self.wc.datatick()

```

Traces(TransmitterProperty) require that three conditions hold for each execution of *pmd_impl.py*.

- $\forall$ *datatick* the total number of preceding *simticks* modulo *const.SIMTICKS PER BIT* = 0.

The `PMD_Transmit.simtick` method defined above is coded to accomplish this condition.

- $\forall$ `writeVota(value)` there is a preceding `pmd_data_request(bit)` and `value = bit`.

`PMD_Transmit.writeVota` is invoked only by the process execution method, `execute` which passes in `value=self.databit`. The only method that updates `self.databit` after initialization is `pmd_data_request`.

```
def execute(self):
    self.wm.registerUserForSimticks(self, True)
    while(True):
        yield passivate, self
        if self.txactive:
            if self.databit != const.NO_DATA:
                self.g(self.increment)
                self.writeVota(self.databit)
```

- $\forall$ `writeVota(value)` the transmitter must be active.

`self.txactive` is initialized to false and the only method that sets it to true is `pmd_tx_start` (above). And since `writeVota` is only invoked by `execute` when `self.txactive` is true then this condition is addressed by the implementation.

In order to satisfy *ReceiverProperty*, the SimPy process `PMD Receive` must implement and expose an external interface consisting of methods named in the signature of *ReceiverProperty* and the traces of its possible executions must be permitted by `Traces(ReceiverProperty)`. The signature, *S*, of *ReceiverProperty* is given by: *S* = {`simtick(simtime)`, `pmd_data_indicate(bit)`, `pmd_cs_indicate(cs)`, `readVota`}. As seen below, `PMD_Receive` has a method for each of these actions whose invocations are traced with an `@traceInvocation` decorator.

```
@traceInvocation
def simtick(self):
    self.numsimticks+=1
```

```

        if self.numsimticks%const.SIMTICKS_PER_BIT==0:
            reactivate(self)

@traceInvocation
def pmd_data_indicate(self, bit):
    self.wc.pmd_data_indicate(bit)

@traceInvocation
def pmd_cs_indicate(self, cs):
    self.wc.pmd_cs_indicate(cs)

@traceInvocation
def readVota(self):
    (response, valuethissimtick)=self.wm.readVota()
    self.valuethissimtick=valuethissimtick
    self.simtickvalues.append(valuethissimtick)

```

Traces(*ReceiverProperty*) require that three conditions hold for each execution of pmd impl.py.

- $\forall$ readVota there is a unique preceding simtick.

readVota is a locally controlled action that is invoked in the process execution method execute which is reactivated by the input action simtick. Thus, each readVota will have a unique preceding simtick.

```

def execute(self):
    self.registerUserForSimticks(self,True)
    while(True):
        yield passivate, self    self.readVota()
        if self.valuethissimtick is const.NO_DATA:
            self.pmd_cs_indicate(False)
        else:
            self.pmd_cs_indicate(True)
        if \
self.numsimticks%const.SIMTICKS_PER_BIT==0:
            if self.numsimticks % \
const.SIMTICKS_PER_BIT ==0:
                self.calculateDataBit()
                self.handleDataBit()
                self.simtickvalues=[]

```

- $\forall$ pmd_cs_indicate(cs) there is a unique preceding readVota. It is clear from execute that this condition is addressed.

- $\forall$ pmd_data_indicate(bit), bit = ZERO or bit = ONE. pmd_data_indicate(bit)

is an output action that is invoked by execute via a helper function handleDataBit which is coded to ensure this condition.

```
def handleDataBit(self):
    if self.databit==const.ZERO or self.databit \
        ==const.ONE:
        self.pmd_data_indicate(self.databit)
```

Physical Medium Dependency Verification

The module pmd_verify.py asserts that the trace properties *TransmitterProperty* and *ReceiverProperty* are true on the traces of an execution of the processes PMD_Transmit and PMD_Receive, as implemented in module pmd_impl.py. It does this by defining VerifyPMDTraces(unittest.TestCase). As an instance of unittest.TestCase, VerifyPMDTraces can be installed in a unittest.TestSuite and run by a unittest.TextTestRunner. It defines a single testPmdTraces method that gets invoked by unittest.TestCase framework. testPmdTraces first verifies the *TransmitterProperty* was not violated by any of the PMD Transmit objects in the simulation. It then verifies that *ReceiverProperty* was not violated by any of the PMD_Receive objects in the simulation. testPmdTraces verifies that trace property *TransmitterProperty* was not violated during the execution of pmd_impl.py by examining the traces of each PMD_Transmit object. For each transmitter, the trace lines are collected and the unittest.TestCase method failUnless is invoked on doTestTransmitterLines, passing in the trace lines for that transmitter. doTestTransmitterLines verifies that each of the conditions required by *TransmitterProperty* are satisfied by invoking in turn the boolean helper methods writeVotaWhenActive, writeVotaWithLastRequestValue and dataTicksAtRightTimes. If

any of these methods return false, `doTestTransmitterLines` immediately returns a false and the unit test will fail.

- *testPmdTraces* verifies that trace property *ReceiverProperty* was not violated during the execution of `pmd impl.py` by examining the traces of each PMD Receive object. For each receiver the trace lines are collected and `unittest.TestCase` method `failUnless` is invoked on *doTestReceiverLines* passing in the trace lines for that receiver `doTestReceiverLines` verifies that each of the conditions required by *ReceiverProperty* are satisfied by invoking in turn the boolean helper methods `simtickReadVotaAlternate`, `readVotaCSAlternate`, `dataIndicateBitsAreValid` and `dataticksAtRightTimes`. If any of these methods return false then `doTestReceiverLines` immediately returns a false and the unit test will fail.

Physical Medium Dependency Test

The test module `pmd_test.py` drives the simulation and verification of the physical medium dependency layer. It defines the necessary components to utilize the PMD, establishes the parameters for the simulation, executes the simulation and then creates and runs a `unittest.TestSuite` based on class `VerifyPmdTraces` defined in the `pmd_verify.py` module.

In order to drive the transmit functionality of the PMD `pmd test.py` defines a class called `TransmitterWirelessChannel`.

```
class TransmitterWirelessChannel(Process):
    def __init__(self):
        Process.__init__(self)
        self.data=[]
        for x in range(simulationLength):
            self.data.append(rand.randrange(0, 3, 1))
        self.index = (-1)
```



```

def setpmdtransmitter(self, pmd):
    self.pmd=pmd

def execute(self):
    while(True):
        yield passivate, self
        if self.index > len(self.data):
            pass
        elif self.index == (-1):
            self.pmd_tx_start()
        elif self.index < len(self.data):
            self.pmd_data_request \
                (self.data[self.index])
        else:
            self.pmd_tx_end()
        self.index+=1

    def datatick(self):
        reactivate(self)

def pmd_data_request(self, bit):
    self.pmd.pmd_data_request(bit)

def pmd_tx_start(self):
    self.pmd.pmd_tx_start()

def pmd_tx_end(self):
    self.pmd.pmd_tx_end()

```

Note first that it is a SimPy process and has the requisite PEM, in this case named `execute`. During initialization, it establishes a random set of data that it will be transmitting throughout the simulation. This data is taken from the set `{const.ZERO, const.ONE, const.NO DATA}`. Within `execute` the locally controlled actions, `pmd_tx_start`, `pmd_tx_end` and `pmd_data_request(bit)`, named for their PMD level counterparts which they invoke, are partitioned and executed based on the preconditions on the state variables. The input action `datatick` performs the function that `simtick` does at the PMD layer, waking the process up.

The receive functionality in `pmd_test.py` is passive. It simply provides a sink for the output actions of the `PMD_Receive` process. It is implemented by `ReceiverWirelessChannel`.

```
class ReceiverWirelessChannel:
    def pmd_data_indicate(self, bit): pass
    def pmd_cs_indicate(self, cs): pass
    def setpmdreceiver(self, pmd): self.pmd=pmd
```

Besides defining the classes that make use of the PMD implementation layer, `pmd_test.py` sets up the parameters for the simulation. A random number (1-5) of transmitters and receivers are determined as well as a random simulation length.

```
from random import Random
rand=Random()
maxNumTransmitters=5
maxNumReceivers=5
numTransmitters=rand.randrange(1,maxNumTransmitters,1)
numReceivers=rand.randrange(1,maxNumReceivers,1)
maxSimulationLength=20000
minSimulationLength=2000
simulationLength=rand.randrange \
    (minSimulationLength, maxSimulationLength,1)
```

With the users of the PMD layer defined and the parameters of the simulation established the necessary objects for the simulation can be created and the simulation executed.

```
def runSim():
    from wm_impl import wm
    initialize()
    activate(wm, wm.simTicker(), prior=True)
    for i in range(numTransmitters):
        twc=TransmitterWirelessChannel()
        tpmd=PMD_Transmit(twc)
        activate(tpmd, tpmd.execute())
        activate(twc, twc.execute())
    for i in range(numReceivers):
        rwc=ReceiverWirelessChannel()
        rpmd=PMD_Receive(rwc)
        activate(rpmd, rpmd.execute())
    simulate(until=simulationLength)
```

```
flushTrace()
runSim()
```

With simulation complete and the traces flushed the execution of the simulation can be verified.

```
from pmd_verify import *
suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(VerifyPMDTraces))
unittest.TextTestRunner(verbosity=2).run(suite)
```

Typical output from an execution of the PMD layer simulation looks like:

```
C:\ms\thesis\gold\pmd>python pmd_test.py
numTransmitters=4 numReceivers=1 simLength=9467
testPmdTraces (pmd_verify.VerifyPMDTraces) ... ok
Ran 1 test in 12.640s OK
```

5.3 Wireless Channel Implementation

The wireless channel layer provides an implementation of an asynchronous shared memory model. It does so by composing `PMD_Transmit` and `PMD_Receive` from the the physical medium dependency layer and providing input and output actions that map to the ports of those two processes. A wireless channel is a way for a station to send and receive bits of data through a single object. Although in this effort there is only a single wireless channel per station it is possible to define multiple wireless channels so that the effect of sending multiple pieces of information simultaneously could be studied.

Wireless Channel Implementation

The functionality of the wireless channel layer is provided by `wc_impl.py`. It must satisfy the *WirelessChannelProperty* by exposing methods that match its signature and permitting only executions that do not violate its traces. The actions in the signature and the trace properties that involve those actions can be partitioned into two groups.

- Those actions that are output actions of the PMD layer and input actions to the wireless channel layer and their corresponding output actions from the wireless channel.

`datatick()` and `wc_datatick()`

- `pmd_data_indicate(bit)` and `wc_data_indicate(bit)`
- `pmd_cs_indicate(cs)` and `wc_cs_indicate(cs)`

Correct execution involving these actions simply means, and the trace property makes explicit, that the actions on the left result in the actions on the right.

```
@traceInvocation
def datatick(self):
    self.wc_datatick()

@traceInvocation
def pmd_data_indicate(self, bit):
    self.wc_data_indicate(bit)

@traceInvocation
def pmd_cs_indicate(self, cs):
    self.wc_cs_indicate(cs)
```

- Those actions that are input actions to the wireless channel from higher layers and their corresponding output actions to the PMD layer.

- `wc_tx_data_request(bit)` and `pmd_data_request(bit)`
- `wc_tx_start()` and `pmd_tx_start()`
- `wc_tx_end()` and `pmd_tx_end()`

Correct execution involving these actions simply means, and the trace property makes explicit, that the actions on the left result in the actions on the right.

```
@traceInvocation
def wc_tx_data_request(self, bit):
    self.pmd_data_request(bit)
```

```
@traceInvocation
def wc_tx_start(self):
    self.pmd_tx_start()
```

```
@traceInvocation
def wc_tx_end(self):
    self.pmd_tx_end()
```

Wireless Channel Verification

The module `wc_verify.py` asserts that none of the conditions required by the trace property *WirelessChannelProperty* are violated. It does so by defining a `unittest.TestCase`, `VerifyWirelessChannelTraces`, which defines the required test method as `testWirelessChannelTraces`. The test method defines helper methods, one for each condition that needs to be tested. It collects these helper methods into the method `doTestWCLines` and then executes that against every wireless channel in the simulation. If any of the helper methods fail for any of the wireless channels then the test case fails and the execution did not satisfy the trace property.

```
def doTestWCLines(lines):
    if not doTestDataticks(lines): return False
    if not doTestDataIndicates(lines): return False
    if not doTestCarrierSenseIndicates(lines):
        return False
    if not doTestDataRequests(lines): return False
    if not doTestTxStarts(lines): return False
    if not doTestTxEnds(lines): return False
    return True

for wc in wirelessChannels:
    wcLines=[]
for line in self.traceLines:
    if line.startswith(str(id(wc))):
        wcLines.append(line)
self.failUnless(doTestWCLines(wcLines))
```

Wireless Channel Test

The test module `wc_test.py` drives the simulation and verification of the wireless channel layer. It defines the necessary components to utilize the wireless channel,

establishes the parameters for the simulation, executes the simulation and then creates and runs a unittest.TestSuite based on VerifyWirelessChannelTraces. First the number of wireless channels that will be executed in the simulation gets determined.

```
from random import Random
rand=Random( )
maxNumWC=5
numWC=rand.randrange( 1,maxNumWC,1 )
```

Then a very simple client of the wireless channel called Plcp is defined. It is a SimPy.Process object that provides the necessary interface to interact with a WirelessChannel object, and implements some rudimentary functionality within its own process execution method to drive it. If the state variable *tx* is True then the Plcp object will transmit data, otherwise it will receive data.

```
class Plcp(Process):
    def __init__(self, tx=False):
        Process.__init__(self)
        self.tx=tx
        self.cs=False

    def setWirelessChannel(self, wc):
        self.wc=wc

    def wc_tx_data_request(self, bit):
        self.wc.wc_tx_data_request(bit)

    def wc_tx_start(self):
        self.wc.wc_tx_start()

    def wc_tx_end(self):
        self.wc.wc_tx_end()

    def wc_datatick(self):
        reactivate(self)

    def wc_data_indicate(self, bit):
        pass

    def wc_cs_indicate(self, cs):
        self.cs=cs
```

```

def execute(self):
    while(True):
        yield passivate, self
        if self.tx and not self.cs:
            self.wc_tx_start()
            yield passivate, self
            self.wc_tx_data_request(1)
            yield passivate, self
            self.wc_tx_end()

```

Then SimPy is initialized, the necessary objects are created with the first Plcp becoming the lone transmitter in the simulation, the process execution methods activated and the simulation run. After the simulation is run the traces get flushed and the verification process follows.

```

def runSim():
    from wm_impl import wm
    from pmd_impl import PMD_Receive, PMD_Transmit
    from wc_impl import WirelessChannel
    from trace_decorators import flushTrace
    from wc_verify import wirelessChannels
    initialize()
    activate(wm, wm.simTicker(), prior=True)
    for i in range(numWC):
        if i==0: plcp=Plcp(True)
        else: plcp=Plcp()
        wc=WirelessChannel(plcp)
        pmdtx=PMD_Transmit(wc)
        pmdrx=PMD_Receive(wc)
        activate(pmdtx, pmdtx.execute())
        activate(pmdrx, pmdrx.execute())
        activate(plcp, plcp.execute())
        wirelessChannels.append(wc)
    simulate(until=simulationLength)
    flushTrace()

runSim()
from wc_verify import *
suite = unittest.TestSuite()
suite.addTest \
    (unittest.makeSuite(VerifyWirelessChannelTraces))
unittest.TextTestRunner(verbosity=2).run(suite)

```

5.4 Physical Layer Convergence Procedure Implementation

The Physical Layer Convergence Procedure, PLCP, provides significant additional functionality on top of the layers that have already been specified, implemented and verified. The PLCP layer turns octets which are supplied by the higher layers into bits for transmission, one bit at a time over the wireless channel. It receives bits from the wireless channel, one bit at a time, that it turns into octets which it supplies to the higher layers. The PLCP layer frames the octets that it is transmitting into protocol service data units (PSDU) that allow the synchronization of sending and receiving stations. It accepts commands from higher layers that control the start and end of transmission. It reports channel status to the higher layers and the results of receiving a PSDU. In short, the PLCP layer provides the functionality to transmit and receive byte oriented data streams across the wireless medium from station to station.

Physical Layer Convergence Procedure Implementation

The PLCP layer implementation module is `plcp_impl.py`. It is responsible for implementing the necessary functionality without violating either *PlcpTransmitProperty* or *PlcpReceiveProperty*. There are three classes defined in `plcp impl.py`.

- `PLCP_Transmit` is responsible for transmitting a Physical layer Protocol Data Unit (PPDU). This class, when active, builds and transmits the frame header so that receiving stations can synchronize for reception. Immediately following the frame header is the payload, the PSDU, delivered an octet at a time by the upper level. `PLCP_Transmit` is a `SimPy.Process`.
- `PLCP_Receive` is responsible for receiving a PPDU. When this class is active it synchronizes with transmitting stations, extracts the payload from the PSDU and delivers it an octet at a time to the upper layer. This process also reports the status

of the medium to the upper layers so that algorithms at those levels can determine when it is safe to transmit. PLCP_Receive is a SimPy.Process.

- PLCP is a composition of PLCP_Transmit and PLCP_Receive. Recall from the PLCP specification that the transmit and receive functionalities are mutually exclusive. Therefore, PLCP activates and deactivates PLCP_Transmit and PLCP_Receive based on the commands that it receives from the higher layers.

`sig(PlcpTransmitProperty)` requires that PLCP implementation of the following external actions.

`tx_start_request`

The request for the PLCP to begin a transmission indicates the number of octets that the higher layer wishes to transmit. This results in the transmitter becoming the active component. The transmitter object method that gets invoked updates local state variables. Additionally, the PLCP output action that notifies the wireless medium gets invoked.

```
@traceInvocation
def tx_start_request(self, numOctets):
    self.activeComponent=self.transmitter
    self.transmitter.tx_start_request(numOctets)
    self.receiver.settransmitting(True)
    self.wc_tx_start()

def tx_start_request(self, numOctets):
    self.numPayloadOctets=numOctets
    self.numOctetsSent=0
    self.txStatus=const.RAMP_UP
```

`wc_tx_start`

When the PLCP is commanded to start a transmission it notifies the wireless medium by invoking the appropriate input action on *WirelessMedium*.

```
@traceInvocation
def wc_tx_start(self):
    self.wc.wc_tx_start()
```

tx_start_confirm

When the transmitter object is ready to begin transmitting octets it notifies the PLCP by invoking the following method which causes the higher layers to be notified that it's time to start sending data, an octet at a time.

```
@traceInvocation
def tx_start_confirm(self):
    self.phy.tx_start_confirm()
```

tx_data_request

The request for the PLCP to transmit an octet of data is passed on to the transmitter object method which decomposes the octets into properly ordered bits and stores them locally for subsequent transmission.

```
@traceInvocation
def tx_data_request(self, octet):
    self.transmitter.tx_data_request(octet)

def tx_data_request(self, octet):
    i=0
    bits=[]
    while(i<8):
        bits.append(int(octet)&1)
        octet=octet>>1
        i+=1
    self.octetBuffer=bits
    self.octetBitNum=0
```

tx_end_request

The request for the PLCP to end a transmission is passed on to the transmitter object and to the wireless channel. The transmitter object uses the notification to update a local state variable to a value indicating that the transmission must stop.

```
@traceInvocation
def tx_end_request(self):
    self.transmitter.tx_end_request()
    self.wc_tx_end()

def tx_end_request(self):
    self.txStatus=0xff
```

wc_tx_data_request

As the transmitter object process executes it transmits bits of data which it extracted from the most recent octet that it received from the PLCP. The transmitter object sends those bits through the PLCP object since it provides the external interface for this layer. The PLCP layer, in turn, forwards the bit to the wireless channel layer for transmission down the stack.

```
@traceInvocation
def wc_tx_data_request(self, bit):
    self.wc.wc_tx_data_request(bit)
```

tx_data_confirm

After the final bit of an octet has been sent, the transmitter object notifies the PLCP layer of confirmation that the octet has been sent. This confirmation is passed to the higher layer which should in response send either the next octet or a command to end the transmission.

```
@traceInvocation
def tx_data_confirm(self):
    self.phy.tx_data_confirm()
```

tx_end_confirm

When the end of a transmission has occurred, either because the higher layer commanded it to end or the end of the data has been reached the PLCP sends confirmation to the higher layers. It also switches to receiver mode of operation and commands the wireless channel to stop transmitting.

```
@traceInvocation
def tx_end_confirm(self):
    self.activeComponent=self.receiver
    self.receiver.settransmitting(False)
    self.phy.tx_end_confirm()
    self.wc.wc_tx_end()
```

Physical Layer Convergence Procedure Verification

The `plcp_verify.py` module asserts that none of the conditions specified by the trace properties *PlcpTransmitProperty* and *PlcpReceiveProperty* are violated during an execution of `plcp_impl.py`. The PLCP verification module defines the `VerifyPlcpTraces` class. It is a subclass of `unittest.TestCase` so it can be invoked as part of a `unittest.TestSuite`. `VerifyPlcpTraces` defines a single test method, `testPLCPTraces`. There are helper methods defined to assert each of the required conditions of two trace properties. These helper methods are invoked in turn on the traces of each PLCP object that ran during the simulation.

`doTestRxProperty01`

This helper function asserts that the required condition

- `rx_start_indication` and `rx_end_indication` alternate, beginning with `rx_start_indication`.

was not violated. If the assertion holds true, this method returns `True`. If the assertion does not hold and the condition was violated, this method returns `False`.

`doTestRxProperty02`

This helper function asserts that the required condition

- Between each `rx_start_indication` and `rx_end_indication`, after every eighth `wc_data_indicate` there is an `rx_data_indication` unless there was a carrier loss.

was not violated. If the assertion holds true, this method returns True. If the assertion does not hold and the condition was violated, this method returns False.

doTestRxProperty03

This helper function asserts that the required condition

- For each `cca_indication(cca)` if `cca=const.CCA IDLE` then the previous `wc_cs_indicate(cs)` had `cs=False`. If `cca=const.CCA BUSY` then the previous `wc_cs_indicate(cs)` had `cs=True`.

was not violated. If the assertion holds true, this method returns True. If the assertion does not hold and the condition was violated, this method returns False.

doTestTxProperty01

This helper function asserts that the required condition

- Each `tx_start_request` will be followed by one `wc_tx_start` which will be followed by one `tx_start_confirm` which will be followed by a `tx_end_confirm`.

was not violated. If the assertion holds true, this method returns True. If the assertion does not hold and the condition was violated, this method returns False.

doTestTxProperty02

This helper function asserts that the required condition

- Each `tx_end_request` will be followed by one `wc_tx_end`.

was not violated. If the assertion holds true then method returns True. If the assertion is found to have not held and the condition was violated then this method returns False.

doTestTxProperty03

This helper function asserts that the required condition

- After a tx_end_request there will be no wc_tx_data request until after a tx_start_request.

was not violated. If the assertion holds true, this method returns True. If the assertion does not hold and the condition was violated, this method returns False.

doTestTxProperty04

This helper function asserts that the required condition

- Each tx_data_request will be followed by eight wc_tx_data request unless a tx_end_request is received.

was not violated. If the assertion holds true, this method returns True. If the assertion does not hold and the condition was violated, this method returns False.

doTestTxProperty05

This helper function asserts that the required condition

- Each tx_data_request will be followed by a tx_data_confirm unless a tx_end_request is received.

was not violated. If the assertion holds true, this method returns True. If the assertion does not hold and the condition was violated, this method returns False.

The test module plcp test.py drives the simulation and verification of the physical layer convergence procedure layer. We are interested in testing the functionality of the PLCP layer, that is the ability to repeatedly coordinate and execute the transmission and reception of byte oriented data a bit at a time across a wireless channel.

The number of iterations, how many byte stream transmissions and receptions, need to be determined.

```

from random import Random
rand=Random()
maxSleepTime=50
maxIterations=10 s
leepTime=rand.randrange(20, maxSleepTime, 1)
numIterations=rand.randrange(1, maxIterations, 1)
simulationLength=numIterations*400

```

In an 802.11 system the client of the PLCP layer would be the physical service layer (PHY). So the next thing that plcp test.py is define classes to mimic the necessary functionality of that layer. In order to simplify the implementation of the test module the PHY functionality has been split into a receiver part and a transmitter part. In reality the PHY layer would be a single component. The PHY receiver is passive, it simply provides the necessary interface.

```

class ReceiverPhy(Process):

    def __init__(self):
        Process.__init__(self)

    def execute(self):
        while(True):
            yield passivate, self

    def setPlcp(self, plcp): self.plcp=plcp
    def cca_indication(self, cca): pass
    def rx_start_indication(self, datalength): pass
    def rx_end_indication(self, rxerror): pass
    def rx_data_indication(self, data): pass

```

Physical Layer Convergence Procedure Test

The PHY transmitter is an active component. It makes use of the PLCP layer to intermittently transmit a stream of byte data. It defines that byte stream and then executes a randomly determined number of iterations in which it transmits that byte stream. The specification *PlcpTransmitProperty* to guides its implementation.

```

class TransmitterPhy(Process):

```

```

def __init__(self):
    Process.__init__(self)
    self.data=[0x00,0x11,0x22, 0x33, 0x44, 0x55, \
0x66, 0x77, 0x88, 0x99, 0xAA, 0xBB, 0xCC, \
0xDD, 0xEE, 0xFF]
    self.index = (-1)

def execute(self):
    iterations=0
    while(True):
        if self.index == (-1):
            self.tx_start_request(len(self.data))
            self.index+=1
        elif self.index < len(self.data):
            self.tx_data_request \
            (self.data[self.index])
            self.index+=1
        elif self.index==len(self.data):
            self.tx_end_request()
            self.index+=1
        else:
            i=0
            while(i<sleepTime):
                i+=1
                yield hold, self, 1.0
                iterations+=1
                if iterations < numIterations:
                    self.tx_start_request \
                    (len(self.data))
                    self.index=0
            yield passivate, self

def setPlcp(self, plcp):
    self.plcp=plcp

def tx_data_request(self, octet):
    self.plcp.tx_data_request(octet)

def tx_start_request(self, numOctets):
    self.plcp.tx_start_request(numOctets)

def tx_start_confirm(self):
    reactivate(self)

def tx_data_confirm(self):
    reactivate(self)

```



```

def tx_end_request(self):
    self.plcp.tx_end_request()

def tx_end_confirm(self):
    self.index+=1
    reactivate(self)

def cca_indication(self, cca):
    pass

```

Once parameters of the simulation are determined and the components to drive it are defined then plcp test.py creates the needed implementation components, runs the simulation, flushes the traces and executes the verification procedure.

```

def runSim():
    from wm_impl import wm
    from wc_impl import WirelessChannel
    from pmd_impl import PMD_Transmit, PMD_Receive
    initialize()
    activate(wm, wm.simTicker(), prior=True)
    from plcp_verify import plcps
    for i in range(3):
        if i==0:
            phy=TransmitterPhy()
        else:
            phy=ReceiverPhy()
        plcp=PLCP(phy)
        wc=WirelessChannel(plcp)
        pmdtx=PMD_Transmit(wc)
        pmdrx=PMD_Receive(wc)
        activate(pmdtx, pmdtx.execute())
        activate(pmdrx, pmdrx.execute())
        activate(plcp.transmitter, \
            plcp.transmitter.execute())
        activate(plcp.receiver, plcp.receiver.execute())
        activate(phy, phy.execute())
        plcps.append(plcp)
    simulate(until=simulationLength)
    flushTrace()
    runSim()

from plcp_verify import *
suite = unittest.TestSuite()
suite.addTest(unittest.makeSuite(VerifyPLCPTraces))
unittest.TextTestRunner(verbosity=2).run(suite)

```

CHAPTER VI

DISCUSSION

A methodology has been presented and demonstrated in this thesis. The methodology consists of a set of tools and procedures. The goals established for the methodology were threefold.

1. The methodology had to be end to end. A clear path from specification through implementation and verification needed to be identified. The specification starts with trace properties. The allowable behavior is defined by the set of action sequences allowed by the trace properties. Trace properties serve as the behavioral specification and are linked directly by their compatible signature with an implementation specification, I/O Automata. It was demonstrated how a linked I/O Automata could be developed from a trace property. The key step from implementation specification to executable implementation was discovered by noting that:

- The constructor of a SimPy process can be used to simply and clearly implement the *states* and *start states* of an I/O Automata.
- Trace decorated methods of a SimPy process could clearly and simply implement the external interface of an I/O Automata.
- A SimPy Process Execution Method could clearly and simply implement an I/O Automata's task partition.

The crucial linkage from implementation specification to executable implementation was established. The trace decorators collected the traces of the

execution and these were logged for verification. Verification was accomplished by developing methods that asserted the truth of the specification on traces of the execution. Thus, the methodology did indeed define a clear path from specification through verification. It is an integrated end to end approach.

2. The methodology had to be readily accessible which has a twofold meaning. First the tools had to be freely available. Licensing could not be an issue. Every tool described in the methodology is freely available, either as open source software or as published work with complete description. Second, each tool had to be usable without special skills or training. The specification tools, though rigorous, are all usable with a basic understanding of set theory, no special grammars are required to be learned. The implementation tools are centered around the powerful and easy to use scripting language, Python. Scripting languages are enabling more programmers to do more things just as system languages opened up large application programming to a wider audience when they replaced assembly languages. With a tool set that is readily available, this methodology should be usable by that larger audience of script programmers.

3. The methodology had to be compositional. Network protocols are layered by necessity. The 802.11 protocol has survived, evolved and thrived because it defined the necessary layers to do so. A single MAC is supported by numerous PHY implementations which are comprised of various physical medium, physical medium dependency and physical layer convergence procedure layers. Without a layered structure a protocol cannot adapt to new requirements. In order to reason about each layer and then combine them the methodology had to be compositional. The specification tools in particular had to support compositional reasoning. Trace properties and I/O Automata do

that extremely well. The compositional implementation was a straight-forward composition of the executable versions.

As noted earlier, a model must capture enough elements of the real system that it is modeling so that the specification, implementation and verification of the model establish the viability of the model. For that reason features of a real 802.11 system were modeled at each level. The result has been a system simulation that communicates byte streams from from an independent process to other independent processes. This is the fundamental purpose of the physical service of the 802.11 protocol so it seems safe to conclude that the viability of the model and hence the methodology has been established.

A significant part to of the 802.11 MAC protocol is the the distributed control function (DCF). It requires and makes use of, composes actually, the PHY service that is implemented by the PLCP component that has been developed. The next logical step in this work is to specify, implement and verify the distributed coordination function. The viability of the methodology has been demonstrated, it should be a straight forward effort. It is interesting to note that the distributed control function is, in reality, a distributed algorithm being run independently by all stations executing within an 802.11 wireless network. Recalling that trace properties and I/O Automata were developed to specify asynchronous components distributed systems it is reasonable to ask: *Does the distributed coordination function of the 802.11 protocol capture enough of the elements of a distributed system to establish the methodology presented here as viable for specifying, implementing and verifying distributed systems?*

BIBLIOGRAPHY

- [1] A. Law, W.D. Kelton, *Simulation Modeling and Analysis*, McGraw Hill, New York, NY, 2000
- [2] G. Fishman, *Discrete-Event Simulation: Modeling, Programming and Analysis*, Springer-Verlag, Berlin, DE, 2001.
- [3] J.D Day, H. Zimmerman, *The OSI reference model*, Proceedings of the IEEE, December 1983.
- [4] R. Braden, *Requirements for Internet Hosts – Communication Layers*, Internet Engineering Task Force, October, 1989.
- [5] B. DiVito, *Integrated Methods for Protocol Specification and Verification*, Proceedings of the IFIP WG6.1 Second International Workshop on Protocol Specification, Testing and Verification, 1983.
- [6] J. Bauer, I. Schaefer, T. Toben, B. Westphal, *Specification and Verification of Dynamic Communication Systems*, IEEE Xplore, June, 2006.
- [7] N. Lynch, *Distributed Algorithms*, Morgan Kaufmann, San Mateo, CA, 1996.
- [8] L. Lamport, *Some Thoughts on Specification*, <http://research.microsoft.com/users/lamPort/tla/notes/92-05-05.txt>, May, 1992.
- [9] J.K. Ousterhout, *Scripting: Higher-Level Programming for the 21st Century*, IEEE Computer, March, 1998.
- [10] *Python Programming Language – Official Website*: <http://www.python.org>.
- [11] *About Python*: <http://www.python.org/about/>.

- [12] J.M. Zelle, *Python as a First Language*,
<http://mcs.wartburg.edu/zelle/python/pythonfirst.html>.
- [13] K. Müller, T. Vignaux, *SimPy Homepage*, <http://simpy.sourceforge.net/>.
- [14] N. Matloff *Introduction to Discrete-Event Simulation and the SimPy Language*,
<http://heather.cs.ucdavis.edu/matloff/156/PLN/DESimIntro.pdf>
- [15] M. Simionato, *The decorator module*, <http://www.phyast.pitt.edu/micheles/python/documentation.html>
- [16] *Python logging Module*, <http://docs.python.org/lib/module-logging.html>
- [17] *Python unittest Module*, <http://docs.python.org/lib/module-unittest.html>
- [18] IEEE 802.11TM Wireless Local Area Networks, <http://ieee802.org/11/>
- [19] G. Ennis, IEEE 802.11TM Tutorials, Part 2: Architecture,
<http://www.ieee802.org/11/Tutorial/archit.pdf>

APPENDICES

APPENDIX A

WIRELESS MEDIUM VERIFICATION CODE

```
import unittest
from trace_decorators import *
from wm_impl import wm
from constants import *

class TestWMTraces(unittest.TestCase):

    class VerifyWMTraces(unittest.TestCase):
        def setUp(self): self.traceLines=getTraceLines()

#-----#
# SVT_NumTx trace verification
# Define a trace property P on NumTx traces s.t.
# Sig(P) = {invocation, response} where:
#   invocation e {g(func=increment)}
#   response = (numTx, v) where v is a non-
#               negative integer
# Traces(P)
# 1 invocation and response alternate beginning with
# invocation. Ensures that invocations and responses are
# atomic
# 2 v, the number of transmitters when invocation
# is made, is equal to the number of previous
# increment invocations. Ensures that the current number
# transmitting is correct based on the invocation history
#-----#

    def testNumTxTraces(self):

        def isIncrement(line):
            parts=line.split(".")
            found=False
            for part in parts:
                if found:
                    return part.startswith("increment")
                if part.startswith("g(func=)":
                    found=True
```



```

        return False

# helper
def isResponse(line):
    try:
        line.index("g: ('numTx'")
        return True
    except:
        return False

# helper
def getvFromResponseLine(line):
    lineparts=line.split(": ")
    responseparts=lineparts[1].split(", ")
    valuepart=responseparts[1]
    return int(valuepart[0])

# P.1
def invRspAlt(lines):
    expecting=const.INVOCATION
    for line in lines:
        if expecting==const.INVOCATION:
            try:
                line.index("g(func=")
                expecting=const.RESPONSE
            except(ValueError):
                return False
        elif expecting==const.RESPONSE:
            try:
                line.index("g: ('numTx'")
                expecting=const.INVOCATION
            except(ValueError):
                return False
    return True

# P.2
def vAlwaysCorrect(lines):
    numIncrements=0
    for line in lines:
        if line.startswith(str(id(wm))):
            numIncrements=0
        elif isIncrement(line):
            numIncrements+=1
        elif not isResponse(line):
            raise Exception \
                ("vAlwaysCrorrect-Unrecognized: "\

```

```

        +line)
    else:
        v=getvFromResponseLine(line)
        if v != numIncrements-1:
            return False
    return True

# Extract the appropriate lines from all
# of the trace lines
numTxLines=[]
for line in self.traceLines:
    if line.startswith(str(id(wm.numtx))):
        numTxLines.append(line)

# Fail if P.1 is violated
self.failUnless(invRspAlt(numTxLines))

# Now get the lines needed for P.2.
simNumtxLines=[]
for line in self.traceLines:
    if line.startswith(str(id(wm))+".simtick("):
        simNumtxLines.append(line)
    elif line.startswith(str(id(wm.numtx))):
        simNumtxLines.append(line)
# Fail if P.2 is violated
self.failUnless(vAlwaysCorrect(simNumtxLines))

#-----#
# SVT_Vota trace verification
# Define a "trace property", P, for SVT_Vota.
# * Sig(P) = {invocation, response} where:
#   - invocation e {read, write(v)}
#   - response = {'votaRead', v} ('votaAck', v)}
# where v is a value from prescribed set of
# values
# * Traces(P) = set of all sequences of acts(P)
# such that:
#   1. v e {const.ZERO, const.ONE, const.NO_DATA,
#      const.INDETERMINATE}
#      - Ensures that only valid values are ever
#        on the air
#   2. For all read responses, ('votaRead', v), v
#      equals the value of v in the previous write
#      invocation
#      - Ensures that the values placed on air
#        are maintained

```

```

#-----#
def testVotaTraces(self):

    # helper function
    def lineType(line):
        if line.startswith(str(id(wm))+\
            ".simtick(simtime)":
            return const.SIMTICK
        try:
            line.index("read(")
            return const.INVOCATION
        except(ValueError):
            pass
        try:
            line.index("write(")
            return const.INVOCATION
        except(ValueError):
            pass
        try:
            line.index("read:")
            return const.RESPONSE
        except(ValueError):
            pass
        try:
            line.index("write:")
            return const.RESPONSE
        except(ValueError):
            raise Exception \
                ("testVotaTraces.lineType-Unknown: "\
                +line)

    # helper function
    def isWriteInvocation(line):
        try:
            line.index("write(v=")
            return True
        except(ValueError):
            return False

    # helper function
    def isReadResponse(line):
        try:
            line.index("read: ")
            return True
        except(ValueError):
            return False

```

```

# helper function
def getvFromWriteInvocation(line):
    lineparts=line.split(".")
    valueparts=lineparts[1].split("=")
    valuepart=valueparts[1]
    return int(valuepart[0])

# helper function
def getvFromReadResponse(line):
    lineparts=line.split(": ")
    valueparts=lineparts[1].split(", ")
    valuepart=valueparts[1]
    return int(valuepart[0])

# this function does the verification
def allvValidForSimtick(lines):
    lastv=const.NO_DATA
    for line in lines:
        if lineType(line) == const.INVOCATION:
            if isWriteInvocation(line):
                lastv=getvFromWriteInvocation\
                    (line)
                if not lastv in \
                    [0, 1, const.NO_DATA, \
                     const.INDETERMINATE]:
                    return False
            elif lineType(line) == const.RESPONSE:
                if isReadResponse(line):
                    if getvFromReadResponse(line) \
                        != lastv:
                        return False
    return True

def allvValid(lines):
    i=0
    linesForSimtick=[]
    for line in lines:
        if lineType(line) == const.SIMTICK:
            if i > 0:
                if not allvValidForSimtick \
                    (linesForSimtick):
                    return False
                linesForSimtick=[]
            i+=1
        else:
            linesForSimtick.append(line)
    return True

```

```

votaLines=[]
for line in self.traceLines:
    # we need the wm simtick lines to reset
    if line.startswith(str(id(wm))+\
        ".simtick(simtime)":
        votaLines.append(line)
    if line.startswith(str(id(wm.vota))):
        votaLines.append(line)

# Fail if either P.1 or P.2 are violated
self.failUnless(allvValid(votaLines))

#-----#
# SVT_WirelessMedium trace verification
#
# Define a trace property P on WirelessMedium traces
# s.t.
# * Sig(P) = {invocation, response, 'simtick'}where:
#   - invocation e {g(func=increment), writeVota,
#   readVota}
#   - response = {'numTx', v), ('votaAck', v),
#   ('votaRead', v)}
#   where v e {const.ZERO, const.ONE,
#   const.NO_DATA, const.INDETERMINATE}
# * Traces(P) = set of sequences of acts(P) s.t.:
# 1. All transmitters and receivers that have
# registered for simticks receive them all.
# 2. All transmitters receive simticks before
# receivers receive them.
#   - Ensures that if there's more than a
#   single transmitter at any time that the
#   value on the air is INDETERMINATE. This
#   provides a composed view of SVT_NumTx and
#   and SVT_Vota traces.
#-----#

def testWMTraces(self):
    def usersActivatedInCorrectOrder(lines):
        try:
            transmitters=wm.getTransmitters()
            receivers=wm.getReceivers()
            users=[]
            for t in transmitters:
                users.append(t)
            for r in receivers:
                users.append(r)

```

```

        expectingOutputAction=True
        index=(-1)
        for line in lines:
            if expectingOutputAction:
                try:
                    line.index("simtime")
                    index+=1
                    expectingOutputAction=\
                        False
                except:
                    return False
            else:
                try:
                    user=users[index]
                    userid=str(id(user))

                    # user was activated
                    # when it was supposed
                    # to be?
                    # throw an exception if
                    # userid not in line.
                    line.index(userid)
                    index+=1
                    if index==len(users):
                        expectingOutputActi
                        on=True
                        index=(-1)
                except:
                    return False
        except:
            return False
    return True

wmSimtickLines=[]
for line in self.traceLines:
    if line.startswith(str(id(wm))):
        try:
            line.index("simtick")
            wmSimtickLines.append(line)
        except:
            pass

self.failUnless(usersActivatedInCorrectOrder \
    (wmSimtickLines))

# verify that for each writeVota(v) that if
# numIncrements > 1 that v=const.INDETERMINATE
# This provides a composed view of SVT_Vota and

```

```

# SVT_NumTx.
# Need to collect lines containing:
# - wm.simtick(simtime
# - wm.g(func=increment
# - wm.writeVota=
def vValidWhenMultipleTransmitters(lines):
    def isSimtickLine(line):
        try:
            line.index("simtime")
            return True
        except(ValueError): return False

    def isIncrementLine(line):
        try:
            line.index("increment")
            return True
        except(ValueError): return False

    def isWriteVotaLine(line):
        try:
            line.index("writeVota")
            return True
        except(ValueError): return False

    def getvFromWriteVotaLine(line):
        lineparts=line.split(",")
        valuepart=lineparts[1][1]
        return int(valuepart)

    numTransmitting=0
    for line in lines:
        if isSimtickLine(line):
            numTransmitting=0
        elif isIncrementLine(line):
            numTransmitting+=1
        elif isWriteVotaLine(line):
            if numTransmitting > 1:
                v=getvFromWriteVotaLine(line)
                if v != const.INDETERMINATE:
                    return False

    return True

wmLines=[]
for line in self.traceLines:
    if line.startswith(str(id(wm))):
        try:
            line.index("simtime")
            wmLines.append(line)

```

```

except( ValueError ):
    try:
        line.index( "increment" )
        wmLines.append( line )
    except( ValueError ):
        try:
            line.index\
            ( "writeVota: ( 'votaAck', " )
            wmLines.append( line )
        except( ValueError ): pass

self.failUnless(
vValidWhenMultipleTransmitters( wmLines ) )

```


APPENDIX B

PHYSICAL MEDIUM DEPENDENCY VERIFICATION CODE

```
import unittest
from constants import *
from trace_decorators import *

"""
This class provides the methods for verifying the
PMD traces. PMD traces come in two types, transmitter
traces and receiver traces.

PMD transmitter traces consist of the following methods:
```

```
    Input Actions:
        simtick()
        pmd_data_request(bit)
        pmd_tx_start()
        pmd_tx_end()
```

```
    Output actions:
        writeVota(value)
        datatick()
```

Specification of the correct behavior of pmd transmitter functionality is done using the above Input/Output Actions.

PMD receiver traces consist of the following Input and Output Actions:

```
    Input Actions:
        simtick()

    Output actions:
        readVota()
        pmd_data_indicate(bit)
        pmd_cs_indicate(cs)
```

Specification of the correct behavior of pmd receiver functionality is done using the above Input/Output Actions.

"""

```
class VerifyPMDTraces(unittest.TestCase):
```

```
    def setUp(self):
        self.traceLines=getTraceLines()
```

```
def testPmdTraces(self):
```

"""

Transmitter trace property:

Define a trace property P s.t.

```
* Sig(P) = {simtick, pmd_data_request(bit),
  pmd_tx_start, pmd_tx_end, writeVota(value),
  datatick}
```

```
* traces(P) = all sequences of elements of
  Sig(P) s.t. for each sequence
```

1. ForAll writeVota(value), the transmitter is active, that is, there is a pmd_tx_start that is more recent than the most recent pmd_tx_end invocation.
2. ForAll writeVota(value), value is == to bit from most recent pmd_data_request(bit)
3. ForAll datatick, the total number of preceding simtick % \ const.SIMTICKS_PER_BIT==0.

"""

```
def doTestTransmitterLines(lines):
```

```
    if not writeVotaWhenActive(lines):
        return False
    if not writeVotaWithLastRequestValue(lines):
        return False
    if not dataticksAtRightTimes(lines):
        return False
    return True
```

"""

P.1

"""

```
def writeVotaWhenActive(lines):
```

```
    active=False
    for line in lines:
        if txLineType(line)== \
            const.LINETYPE_PMD_TX_START:
```

```

        active=True
    elif txLineType(line)== \
        const.LINETYPE_PMD_TX_END:
        active=False
    elif txLineType(line)== \
        const.LINETYPE_PMD_WRITE_VOTA:
        if active==False:
            return False
    return True

"""
P.2
"""
def writeVotaWithLastRequestValue(lines):
    lrv=None
    for line in lines:
        if txLineType(line)== \
            const.LINETYPE_PMD_DATA_REQUEST:
            lineparts=line.split("=")
            lrv=lineparts[1][0]
        elif txLineType(line)== \
            const.LINETYPE_PMD_WRITE_VOTA:
            lineparts=line.split("=")
            v=lineparts[1][0]
            if v != lrv:
                return False
    return True

"""
P.3
"""
def dataticksAtRightTimes(lines):
    numsimticks=0
    for line in lines:
        if txLineType(line)== \
            const.LINETYPE_PMD_SIMTICK:
            numsimticks+=1
        elif txLineType(line)== \
            const.LINETYPE_PMD_DATATICK:
            if numsimticks % \
                const.SIMTICKS_PER_BIT != 0:
                return False
    return True

def txLineType(line):
    try:
        line.index("pmd_tx_start(")
        return const.LINETYPE_PMD_TX_START

```

```

except(ValueError):
    pass
try:
    line.index("pmd_tx_end(")
    return const.LINETYPE_PMD_TX_END
except(ValueError):
    pass
try:
    line.index("writeVota(")
    return const.LINETYPE_PMD_WRITE_VOTA
except(ValueError):
    pass
try:
    line.index("pmd_data_request(")
    return const.LINETYPE_PMD_DATA_REQUEST
except(ValueError):
    pass
try:
    line.index("simtick()")
    return const.LINETYPE_PMD_SIMTICK
except(ValueError):
    pass
try:
    line.index("datatick()")
    return const.LINETYPE_PMD_DATATICK
except(ValueError):
    raise Exception \
        ("testPmdTraces-Unknown: "+line)

```

"""-----

Receiver trace property:

Define a trace property P s.t.

- * Sig(P) = {simtick, readVota, pmd_data_indicate(bit), pmd_cs_indicate(cs)}
- * traces(P) = all sequences of elements of Sig(P) s.t. for each sequence
 1. ForAll readVota actions there is a unique preceding simtick action.
 2. ForAll pmd_cs_indicate(cs) actions there is a unique preceding readVota action. If the readVota action returned const.NO_DATA then the value of cs is False, else it is True.
 3. ForAll pmd_data_indicate(bit) actions bit is equal to const.ZERO or const.ONE.
 4. ForAll pmd_data_indicate(bit) actions there are a multiple of

```

        const.SIMTICKS_PER_BIT simtick actions
        since the previous pmd_data_indicate
        action.
-----" " "
def doTestReceiverLines(lines):
    if not simtickReadVotaAlternate(lines):
        return False
    if not readVotaCSAlternate(lines):
        return False
    if not dataIndicateBitsAreValid(lines):
        return False
    if not dataIndicateAtCorrectTimes(lines):
        return False
    return True

" " "
P.1
" " "
def simtickReadVotaAlternate(lines):
    lastType=const.LINETYPE_NONE
    for line in lines:
        lineType=rxLineType(line)
        if lineType==const.LINETYPE_PMD_SIMTICK \
        or lineType==const.LINETYPE_PMD_READ_VOTA:
            if lineType==lastType:
                return False
            lastType=lineType
    return True

" " "
P.2
" " "
def readVotaCSAlternate(lines):
    lastType=const.LINETYPE_NONE
    for line in lines:
        lineType=rxLineType(line)
        if lineType==const.LINETYPE_PMD_READ_VOTA\
        or lineType==\
        const.LINETYPE_PMD_CS_INDICATE:
            if lineType==lastType:
                return False
            lastType=lineType
    return True

" " "

```

P.3

"""

```
def dataIndicateBitsAreValid(lines):
    for line in lines:
        if rxLineType(line) == \
            const.LINETYPE_PMD_DATA_INDICATE:
            lineparts=line.split("=")
            bit=lineparts[1][0]
            if int(bit) != \
                const.ZERO and int(bit) != \
                const.ONE:
                return False
    return True
```

"""

P.4

"""

```
def dataIndicateAtCorrectTimes(lines):
    numsimticks=0
    for line in lines:
        lineType=rxLineType(line)
        if lineType==const.LINETYPE_PMD_SIMTICK:
            numsimticks+=1
        elif lineType== \
            const.LINETYPE_PMD_DATA_INDICATE:
            if numsimticks % \
                const.SIMTICKS_PER_BIT != 0:
                return False
    return True
```

```
def rxLineType(line):
    try:
        line.index("simtick()")
        return const.LINETYPE_PMD_SIMTICK
    except(ValueError):
        pass
    try:
        line.index("readVota()")
        return const.LINETYPE_PMD_READ_VOTA
    except(ValueError):
        pass
    try:
        line.index("pmd_cs_indicate()")
        return const.LINETYPE_PMD_CS_INDICATE
    except(ValueError):
        pass
    try:
        line.index("pmd_data_indicate()")
```

```

        return const.LINETYPE_PMD_DATA_INDICATE
    except(ValueError):
        raise Exception \
            ("testPmdTraces-Unknown: "+line)

from wm_impl import wm

# verify the traces for each of the transmitters
pmd_transmitters=wm.getTransmitters()
for transmitter in pmd_transmitters:
    transmitterLines=[]
    for line in self.traceLines:
        if line.startswith(str(id(transmitter))):
            transmitterLines.append(line)

# Fail if any of the transmitter trace
# properties # are violated.
self.failUnless \
    (doTestTransmitterLines(transmitterLines))

pmd_receivers=wm.getReceivers()
# verify the traces for each of the receivers
for receiver in pmd_receivers:
    receiverLines=[]
    for line in self.traceLines:
        if line.startswith(str(id(receiver))):
            receiverLines.append(line)

# Fail if any receiver trace properties
# are violated.
self.failUnless(doTestReceiverLines \
    (receiverLines))

# made it through the verification process without
# failing
return True

```

APPENDIX C

WIRELESS CHANNEL VERIFICATION CODE

```
"""-----  
This class provides the methods for verifying the  
WirelessChannel traces. WirelessChannel traces consist  
of the following actions:
```

```
    Input Actions from pmd  
        datatick()  
        pmd_data_indicate(bit)  
        pmd_cs_indicate(cs)
```

```
    Output Actions to pmd  
        pmd_data_request(bit)  
        pmd_tx_start()  
        pmd_tx_end()
```

```
    Input Actions from plcp  
        wc_tx_data_request(bit)  
        wc_tx_start()  
        wc_tx_end()
```

```
    Output Actions to plcp  
        wc_datatick()  
        wc_data_indicate(bit)  
        wc_cs_indicate(cs)
```

```
The trace properties that define a correct execution of  
WirelessChannel are:
```

- * datatick() and wc_datatick() alternate, beginning
with datatick()
- * pmd_data_indicate(bit) and wc_data_indicate(bit)
alternate, beginning with pmd_data_indicate(bit).
For each pmd_data_indicate(bit) the following
wc_data_indicate(bit) will have "bit" of the same
value.


```

* pmd_cs_indicate(cs) and wc_cs_indicate(cs) alternate,
  beginning with pmd_cs_indicate(cs).
  For each pmd_cs_indicate(cs) the following
  wc_cs_indicate(cs) will have "cs" of the same value.

* wc_tx_data_request(bit) and pmd_data_request(bit)
  alternate, beginning with wc_tx_data_request(bit).
  For each wc_tx_data_request(bit) the following
  pmd_data_request(bit) will have "bit" of the same
  value.

* wc_tx_start() and pmd_tx_start() alternate, beginning
  with wc_tx_start().

* wc_tx_end() and pmd_tx_end() alternate, beginning
  with wc_tx_end().
"""

import unittest
from constants import *
from trace_decorators import *

# populated by wc_test
wirelessChannels=[]
class VerifyWirelessChannelTraces(unittest.TestCase):

    def setUp(self):
        self.traceLines=getTraceLines()

    def testWirelessChannelTraces(self):

        def wcLineType(line):
            try:
                line.index("wc_datatick(")
                return const.LINETYPE_WC_DATATICK
            except(ValueError):
                pass
            try:
                line.index("datatick(")
                return const.LINETYPE_PMD_DATATICK
            except(ValueError):
                pass
            try:
                line.index("pmd_data_indicate(")
                return const.LINETYPE_PMD_DATA_INDICATE
            except(ValueError):
                pass

```

```

try:
    line.index("wc_data_indicate(")
    return const.LINETYPE_WC_DATA_INDICATE
except(ValueError):
    pass
try:
    line.index("pmd_cs_indicate(")
    return const.LINETYPE_PMD_CS_INDICATE
except(ValueError):
    pass
try:
    line.index("wc_cs_indicate(")
    return const.LINETYPE_WC_CS_INDICATE
except(ValueError):
    pass
try:
    line.index("wc_tx_data_request(")
    return\
    const.LINETYPE_WC_TX_DATA_REQUEST
except(ValueError):
    pass
try:
    line.index("pmd_data_request(")
    return const.LINETYPE_PMD_DATA_REQUEST
except(ValueError):
    pass
try:
    line.index("wc_tx_start(")
    return const.LINETYPE_WC_TX_START
except(ValueError):
    pass
try:
    line.index("pmd_tx_start(")
    return const.LINETYPE_PMD_TX_START
except(ValueError):
    pass
try:
    line.index("wc_tx_end(")
    return const.LINETYPE_WC_TX_END
except(ValueError):
    pass
try:
    line.index("pmd_tx_end(")
    return const.LINETYPE_PMD_TX_END
except(ValueError):
    raise Exception
("VerifyWirelessChannelTraces " +line)

```

```

def doTestDataticks(lines):
    expectingPMD=True
    for line in lines:
        if wcLineType(line)== \
            const.LINETYPE_WC_DATATICK:
            if expectingPMD:
                return False
            else:
                expectingPMD=True
        elif wcLineType(line)== \
            const.LINETYPE_PMD_DATATICK:
            if not expectingPMD:
                return False
            else:
                expectingPMD=False
    return True

def doTestDataIndicates(lines):
    expectingPMD=True
    lastBit=const.NO_DATA
    for line in lines:
        if wcLineType(line)== \
            const.LINETYPE_WC_DATA_INDICATE:
            if expectingPMD:
                return False
            else:
                if getValueFromLine(line)== \
                    lastBit:
                    expectingPMD=True
                else:
                    return False
        elif wcLineType(line)== \
            const.LINETYPE_PMD_DATA_INDICATE:
            if not expectingPMD:
                return False
            else:
                expectingPMD=False
                lastBit=getValueFromLine(line)
    return True

def getValueFromLine(line):
    lineparts=line.split("=")
    value=lineparts[1].split(" ")[0]
    return value

def doTestCarrierSenseIndicates(lines):
    expectingPMD=True
    lastCS=False

```

```

for line in lines:
    if wcLineType(line)== \
    const.LINETYPE_WC_CS_INDICATE:
        if expectingPMD:
            return False
        else:
            if getValueFromLine(line)==lastCS:
                expectingPMD=True
            else:
                return False
    elif wcLineType(line)== \
    const.LINETYPE_PMD_CS_INDICATE:
        if not expectingPMD:
            return False
        else:
            expectingPMD=False
            lastCS=getValueFromLine(line)
return True

def doTestDataRequests(lines):
    expectingWC=True
    lastBit=const.NO_DATA
    for line in lines:
        if wcLineType(line)== \
        const.LINETYPE_PMD_DATA_REQUEST:
            if expectingWC:
                return False
            else:
                if getValueFromLine(line)== \
                lastBit:
                    expectingWC=True
                else:
                    return False
        elif wcLineType(line)== \
        const.LINETYPE_WC_TX_DATA_REQUEST:
            if not expectingWC:
                return False
            else:
                expectingWC=False
                lastBit=getValueFromLine(line)
    return True

def doTestTxStarts(lines):
    expectingWC=True
    for line in lines:
        if wcLineType(line)== \
        const.LINETYPE_PMD_TX_START:
            if expectingWC:

```

```

        return False
    else:
        expectingWC=True
    elif wcLineType(line)== \
const.LINETYPE_WC_TX_START:
        if not expectingWC:
            return False
        else:
            expectingWC=False
    return True

def doTestTxEnds(lines):
    expectingWC=True
    for line in lines:
        if wcLineType(line)== \
const.LINETYPE_PMD_TX_END:
            if expectingWC:
                return False
            else:
                expectingWC=True
        elif wcLineType(line)== \
const.LINETYPE_WC_TX_END:
            if not expectingWC:
                return False
            else:
                expectingWC=False
    return True

def doTestWCLines(lines):
    if not doTestDataticks(lines): return False
    if not doTestDataIndicates(lines):
        return False
    if not doTestCarrierSenseIndicates(lines):
        return False
    if not doTestDataRequests(lines): return False
    if not doTestTxStarts(lines): return False
    if not doTestTxEnds(lines): return False
    return True

for wc in wirelessChannels:
    wcLines=[]
    for line in self.traceLines:
        if line.startswith(str(id(wc))):
            wcLines.append(line)

self.failUnless(doTestWCLines(wcLines))

```

APPENDIX D

PHYSICAL LAYER CONVERGENCE PROCEDURE VERIFICATION CODE

```

" " "-----
Interface to wc

    Output Actions to wc
        * wc_tx_data_request(bit)
        * wc_tx_start()
        * wc_tx_end()

    Input Actions from wc
        * wc_datatick()
        * wc_data_indicate(bit)
        * wc_cs_indicate(cs)

Interface to phy for transmission:

    Input actions
        * tx_data_request(octet)
        * tx_start_request(numOctets)
        * tx_end_request()

    Output actions
        * tx_start_confirm()
        * tx_end_confirm()
        * tx_data_confirm()

Interface to phy for reception:

    Output Actions
        * cca_indication(cca)
        * rx_start_indication(datalength)
        * rx_data_indication(data)
        * rx_end_indication(rxerror)

Receive:
-----

RxProperty01
rx_start_indication(datalength) and
```

rx_end_indication (rxerror) alternate,
beginning with rx_start_indication.

RxProperty02

Between each rx_start_indication and rx_end_indication,
after every eighth wc_data_indicate there is an
rx_data_indication unless there was a carrier loss.

RxProperty03

For each cca_indication(cca) if cca=const.CCA_IDLE then
the previous wc_cs_indicate(cs) had cs=False.
If cca=const.CCA_BUSY then the previous
wc_cs_indicate(cs) had cs=True

Transmit:

TxProperty01

Each tx_start_request will be followed by one wc_tx_start
which will be followed by one tx_start_confirm which will
be followed by a tx_end_confirm.

TxProperty02

Each tx_end_request will be followed by one wc_tx_end

TxProperty03

After a tx_end_request there will be no wc_tx_data_request
until after a tx_start_request.

TxProperty04

Each tx_data_request will be followed by eight
wc_tx_data_request unless a tx_end_request is received

TxProperty05

Each tx_data_request will be followed by a tx_data_confirm
unless a tx_end_request is received

"""

```
import unittest
from constants import *
from trace_decorators import *
```

```
# populated by plcp_test
plcps=[]
```

```
class VerifyPLCPTraces(unittest.TestCase):
    def setUp(self):
        self.traceLines=getTraceLines()
```

```

def testPLCPTraces(self):

    def plcpLineType(line):
        try:
            line.index("wc_tx_data_request(")
            return \
                const.LINETYPE_WC_TX_DATA_REQUEST
        except(ValueError):
            pass
        try:
            line.index("wc_tx_start(")
            return const.LINETYPE_WC_TX_START
        except(ValueError):
            pass
        try:
            line.index("wc_tx_end(")
            return const.LINETYPE_WC_TX_END
        except(ValueError):
            pass
        try:
            line.index("wc_data_indicate(")
            return const.LINETYPE_WC_DATA_INDICATE
        except(ValueError):
            pass
        try:
            line.index("wc_cs_indicate(")
            return const.LINETYPE_WC_CS_INDICATE
        except(ValueError):
            pass
        try:
            line.index("tx_data_request(")
            return \
                const.LINETYPE_PLCP_TX_DATA_REQUEST
        except(ValueError):
            pass
        try:
            line.index("tx_start_request(")
            return \
                const.LINETYPE_PLCP_TX_START_REQUEST
        except(ValueError):
            pass
        try:
            line.index("tx_end_request(")
            return \
                const.LINETYPE_PLCP_TX_END_REQUEST

```



```

except(ValueError):
    pass
try:
    line.index("tx_start_confirm(")
    return \
        const.LINETYPE_PLCP_TX_START_CONFIRM
except(ValueError):
    pass
try:
    line.index("tx_end_confirm(")
    return \
        const.LINETYPE_PLCP_TX_END_CONFIRM
except(ValueError):
    pass
try:
    line.index("tx_data_confirm(")
    return \
        const.LINETYPE_PLCP_TX_DATA_CONFIRM
except(ValueError):
    pass
try:
    line.index("cca_indication(")
    return \
        const.LINETYPE_PLCP_CCA_INDICATION
except(ValueError):
    pass
try:
    line.index("rx_start_indication(")
    return \
        const.LINETYPE_PLCP_RX_START_INDICATION
except(ValueError):
    pass
try:
    line.index("rx_data_indication(")
    return \
        const.LINETYPE_PLCP_RX_DATA_INDICATION
except(ValueError):
    pass
try:
    line.index("rx_end_indication(")
    return \
        const.LINETYPE_PLCP_RX_END_INDICATION
except(ValueError):
    pass
try:
    line.index("wc_datatick(")
    return const.LINETYPE_WC_DATATICK

```

```

except(ValueError):
    raise Exception \
        ("testPLCPTraces-Unknown: "+line)

def doTestRxProperty01(lines):
    expectingSTART=True
    for line in lines:
        if plcpLineType(line)== \
            const.LINETYPE_PLCP_RX_END_INDICATION:
            if expectingSTART:
                return False
            else:
                expectingSTART=True
        elif plcpLineType(line)== \
            const.LINETYPE_PLCP_RX_START_INDICATION:
            if not expectingSTART:
                return False
            else:
                expectingSTART=False
    return True

def doTestRxProperty02(lines):
    start=0
    wdata=1
    rxdata=2
    numwcind=0
    cs=True
    expecting=start
    for line in lines:
        lineType=plcpLineType(line)
        if lineType== \
            const.LINETYPE_PLCP_RX_START_INDICATION:
            numwcind=0
            if expecting==start:
                expecting=wdata
            else:
                return False
        elif lineType== \
            const.LINETYPE_PLCP_RX_END_INDICATION:
            if expecting!=start:
                expecting=start
            else:
                return False
        elif lineType== \

```

```

const.LINETYPE_WC_DATA_INDICATE:
    numwcind+=1
    if expecting==wdata and numwcind==8:
        expecting=rxdata
        numwcind=0
elif lineType== \
const.LINETYPE_PLCP_RX_DATA_INDICATION:
    if expecting!=rxdata and cs==True:
        return False
    else:
        expecting=wdata
        cs=True
elif lineType== \
const.LINETYPE_WC_CS_INDICATE:
    thiscs=getValueFromLine(line)
    if expecting==wdata:
        if thiscs=="False":
            cs=False
return True

```

```

def doTestRxProperty03(lines):
    lastWCCSValue=None
    for line in lines:
        if plcpLineType(line)== \
const.LINETYPE_PLCP_CCA_INDICATION:
            ccaValue=getValueFromLine(line)
            if ccaValue==const.CCA_IDLE:
                if lastWCCSValue == "True":
                    return False
            elif ccaValue==const.CCA_BUSY:
                if lastWCCSValue == "False":
                    return False
        elif plcpLineType(line)== \
const.LINETYPE_WC_CS_INDICATE:
            lastWCCSValue=getValueFromLine(line)
    return True

```

```

def doTestTxProperty01(lines):
    startrequest=0
    wcstart=1
    startconfirm=2
    endconfirm=3
    expecting=startrequest
    for line in lines:

```

```

lineType=plcpLineType(line)
if lineType== \
const.LINETYPE_PLCP_TX_START_REQUEST:
    if expecting!=startrequest:
        return False
    else:
        expecting=wcstart
elif lineType==const.LINETYPE_WC_TX_START:
    if expecting!=wcstart:
        return False
    else:
        expecting=startconfirm
elif lineType== \
const.LINETYPE_PLCP_TX_START_CONFIRM:
    if expecting!=startconfirm:
        return False
    else:
        expecting=endconfirm
elif lineType== \
const.LINETYPE_PLCP_TX_END_CONFIRM:
    if expecting!=endconfirm:
        return False
    else:
        expecting=startrequest
return True

```

```

def doTestTxProperty02(lines):
    endrequest=0
    wcend=1
    expecting=endrequest
    for line in lines:
        lineType=plcpLineType(line)
        if lineType== \
const.LINETYPE_PLCP_TX_END_REQUEST:
            if expecting!=endrequest:
                return False
            else:
                expecting=wcend
        elif lineType==const.LINETYPE_WC_TX_END:
            if expecting!=wcend:
                return False
            else:
                expecting=endrequest
    return True

```

```

def doTestTxProperty03(lines):
    forbidden=True
    for line in lines:
        lineType=plcpLineType(line)
        if lineType== \
            const.LINETYPE_PLCP_TX_END_REQUEST:
            forbidden=True
        if lineType== \
            const.LINETYPE_WC_TX_DATA_REQUEST:
            if forbidden:
                return False
        if lineType== \
            const.LINETYPE_PLCP_TX_START_REQUEST:
            forbidden=False
    return True

def doTestTxProperty04(lines):
    dataStarted=False
    plcprequest=0
    wcrequest=1
    numwcrequest=0
    expecting=plcprequest
    for line in lines:
        lineType=plcpLineType(line)
        if lineType== \
            const.LINETYPE_PLCP_TX_DATA_REQUEST:
            if expecting!=plcprequest:
                return False
            else:
                dataStarted=True
                expecting=wcrequest
        elif lineType== \
            const.LINETYPE_WC_TX_DATA_REQUEST:
            if not dataStarted:
                continue
            elif expecting!=wcrequest:
                return False
            else:
                numwcrequest+=1
                if numwcrequest==8:
                    expecting=plcprequest
                    numwcrequest=0
        elif lineType== \
            const.LINETYPE_PLCP_TX_END_REQUEST:
            expecting=plcprequest
            numwcrequest=0

```

```

        dataStarted=False
    return True

```

```

def doTestTxProperty05(lines):
    datarequest=0
    dataconfirm=1
    endrequest=2
    expecting=datarequest
    for line in lines:
        lineType=plcpLineType(line)
        if lineType== \
            const.LINETYPE_PLCP_TX_DATA_REQUEST:
            if expecting!=datarequest:
                return False
            else:
                expecting=dataconfirm
        if lineType== \
            const.LINETYPE_PLCP_TX_DATA_CONFIRM:
            if expecting!=dataconfirm:
                return False
            else:
                expecting=datarequest
        if lineType== \
            const.LINETYPE_PLCP_TX_END_REQUEST:
            expecting=datarequest
    return True

```

```

def getValueFromLine(line):
    lineparts=line.split("=")
    value=lineparts[1].split(" ")[0]
    return value

```

```

def doTestPLCPLines(lines):
    if not doTestRxProperty01(lines): return False
    if not doTestRxProperty02(lines): return False
    if not doTestRxProperty03(lines): return False
    if not doTestTxProperty01(lines): return False
    if not doTestTxProperty02(lines): return False
    if not doTestTxProperty03(lines): return False
    if not doTestTxProperty04(lines): return False
    if not doTestTxProperty05(lines): return False
    return True

```

```
for plcp in plcps:
    plcpLines=[]
    for line in self.traceLines:
        if line.startswith(str(id(plcp))):
            plcpLines.append(line)

self.failUnless(doTestPLCPLines(plcpLines))
```