Proceedings of the 2006 Winter Simulation Conference
L. F. Perrone, F. P. Wieland, J. Liu, B. G. Lawson, D. M. Nicol, and R. M. Fujimoto, eds.

WIRELESS NETWORK SIMULATION EXTENSIONS IN SIDE/SMURPH

Pawel Gburzynski
Ioanis Nikolaidis

Computing Science Department
University of Alberta
Edmonton, AB T6G 2E8, Canada

ABSTRACT

We describe the most recent step in the evolution of SIDE/SMURPH and, specifically, a generic model of a wireless channel, which enables to use the package for accurately modeling wireless networks, especially ad–hoc networks consisting of a potentially large number of possibly mobile nodes. The generic nature of the channel model allows the user to introduce functions describing the propagation characteristics of the actual wireless medium, e.g., the impact of distance on signal level and interference, as well as the relationship between the signal-to-interference ratio and the probability of a successful packet reception. To illustrate the capabilities of the supported extensions, we review an example of a shadowing channel model.

1 INTRODUCTION

Producing high fidelity simulations of wireless networks, such as mobile wireless ad hoc networks or wireless sensor networks is a concern of many researchers. Yet, as (Kotz et al. 2004) pointed out, the vast majority of simulations adopt dangerously simplifying "axioms" when modeling wireless environments. The end result is that the research community, instead of using simulations to express realistic scenarios, routinely uses simplifications of the kind one would expect in analytical studies. The approach is quite contrary to the philosophy of why simulations are supposed to be built in the first place, i.e., to study a rich set of realistic scenarios, often well beyond what analytical techniques can capture. In addition, it was also noted (Haq and Kunz 2005) that significant discrepancy between results from two popular simulators (ns-2 and GlomoSim) exists against each other as well as against an emulated network. Thus, there is evidence that simulators are still not always capturing accurately the physical layer, and equally importantly, the *network–wide* performance results are *not* insensitive to the precision of the wireless channel model.

While it is true that our understanding of physical channels evolves, it does not remove the burden for simulators to provide suitable "hooks" where a user can *easily* tinker and correct the simulated physical model and bring it as close as possible to the behavior of the real system. To make matters worse, recent advances in wireless network in the area of *cross–layer protocol designs* have further compounded the demands put on wireless network simulations. Cross–layer designs relax, or even dissolve, the layered model in the hope of better performance by trying to capitalize on the opportunities afforded by the physical layer.

Clearly, simulation environments that separate the layers, either in the interest of reducing programming complexity, or just for the sake of modularity, are an unnatural match for cross–layer design. The physical layer is no longer a simple "bit pipe", and needs to be accounted for as a time– and space–variable, possibly "intelligent", sub–system that deserves to be explicitly modeled. In fact the dynamics of the physical channel can interact with temporal aspects of higher layer protocol (e.g., transmissions of control messages) and hence the entire system's performance is determined by interactions that could not have been expressed by simple models where the physical layer is represented as a sequence of random coin flips to express a certain bit error rate.

This is not to say that cross–layer design is the only reason why better wireless channel simulations are needed. Consider for example the case of Hybrid ARQ (HARQ) schemes. For example, certain Type–II HARQ schemes combine the received "soft" values per symbol over successive (re)transmissions of a frame (each one subjected to different fading) and using an algorithm, such as Maximum Ratio Combining (MRC), to extract the correct "hard" data. Contrary to what most simulators would be able to express, multiple packet (re)transmissions are combined into forming one packet arrival at the receiver. Other examples of diversity (spatial or temporal) exist where the finer study of the physical layer is significant, e.g., Multiple Input Multiple Output (MIMO) systems, collaborative interference cancellation schemes, etc.

The rest of the paper presents the wireless extensions recently incorporated into SIDE/SMURPH and the rationale

behind them. SIDE/SMURPH has been described in detail elsewhere (Gburzynski 1996) and follows a long evolution since the 80s (under various names: LANSF, SMURPH, SIDE). It is a discrete event simulation environment and in its current form it includes a generic model for wireless channel support. In principle, all extensible simulation environments can be extended to eventually include better support for wireless simulations. What we hope is achieved with the SIDE/SMURPH extensions is that the right kind of abstractions have been selected, to encourage the incorporation of more accurate physical channel models without unduly hurting the ability to develop simulations quickly. In other simulation packages, e.g., ns-2, the reluctance of researchers to tailor the physical layer models, suggests that the task is perceived as being rather cumbersome. Similar concerns appear to hold for other systems, even if some of them, e.g., GloMoSim, incorporated more refined wireless models to begin with.

The chosen abstractions express what we believe are useful features for describing the behavior of the wireless channel. Certainly, only time will tell whether the presented extensions are successful, and in fact they may need to evolve even further. We believe that providing the right abstractions is instrumental to encouraging users to think in particular terms about the problem at hand. To make the point that the presented extensions are not onerous to use, we provide a short example of a shadowing channel model.

The remainder of the paper is organized as follows: section 2 outlines the two basic abstractions used in the modeling extensions implemented in SIDE/SMURPH for wireless network simulation support. Certain basic concepts (the "interference histogram", and the "activity assessment") used by the model are also introduced. Section 3 is a short example illustrating the expressive power of the extensions. Section 4 concludes the paper.

2 THE DESIGN OF WIRELESS EXTENSIONS

2.1 Concepts

SIDE/SMURPH already supports abstractions for *links* and *ports*, describing, respectively, (broadcast) transmission media and points of attachments of nodes/stations to such media. Transmissions are received via ports after a certain propagation delay. The use of the same abstractions for wireless environments presented several shortcomings. First, in a wireless environment, nodes/interfaces attached, in a logical sense, to a link vary over time with changing distances due to mobility and physical phenomena, e.g., fading, thus forming time–varying "neighborhoods". The temptation to use one single link connecting all nodes and to, *post–facto*, i.e., after delivering the events to the nodes, decide which events (such as the beginning of a packet reception) would be visible to which nodes would result in significant scalability problems. Many events would have had to be ignored *after* having already burdened the simulator with overhead for their processing.

We introduce to SIDE/SMURPH a new abstraction, called an `RFChannel`, to model wireless links. `RFChannel` influences which events (if any) will be delivered to each node. It models the channel and hence the events it generates depend on the exact conditions of the channel, including of course distances between transmitting and receiving node but also arbitrary functions that influence the signal strength that can describe position- and time–dependent phenomena, such as fading and shadowing.

The original *port* abstraction presented also limited flexibility, especially when deciding whether a received signal would eventually morph into a received packet or not. Again, one could entertain the idea of post–facto calculation of whether particular conditions were met during the reception of a packet to decide if it was really received or not, but at the detriment of expressiveness, generality, and most of all, efficiency.

As an alternative to ports, we introduce the `Transceiver` abstraction which essentially defines a point in space (2D or 3D) at which the combined received signal of one or more transmissions could trigger events that signify packet activity. There is no guarantee that events at the transceiver will eventually result in a packet being received at a node. Hence, it becomes possible to express complicated conditions that govern the acceptability of a packet, for example how many symbol errors in sequence could render the received signal undecodable. In essence, transceivers introduce the potential to express relative merits of wireless receiver structures, beyond just the detection threshold (which is also modeled).

From the viewpoint of a receiver, the potential to receive and correctly decode a packet is being re-assessed, at least in principle, at any point in time. However, the per–symbol simulation of a channel would have been extremely inefficient for a large network. Instead we opted to separate the early (crucial) preamble part of a packet (without its acquisition a receiver cannot "follow" through the rest of the transmission) and the latter part which incorporates the payload. A separation between a preamble and main body is found in all wireless transmission protocols and it is a requirement for receiver synchronization. In addition to noticing the preamble separate and earlier than the rest of the packet (and applying to it different criteria to determine if it is acceptable), we also retain the history of what was the impact of other signals on the received signal in order to re–assess the quality of the reception as needed. That is, even when the events related to the preamble and to the rest of the packet are delivered, the history of what interference occurred *during* the preamble and during the main body of the packet are still accessible to the programmer.

In summary, at a receiver, the events resulting from a transmission are not at an exhausting level (one event per

transmitted symbol per packet) but just enough to relate to the basic functions expected of a receiver, i.e., the decision to acquire or not a transmission based on received preamble and (if it decides to acquire the signal) the process of following through with the reception until either the end of the transmission or up to a point where a particular condition, such as low signal strength, is met that forces aborting the reception. The transceiver is of course capable of inspecting the interference histogram and deciding on even more specific issues, for example whether the error correction scheme could be applied successfully or not, etc., which is useful if interested in collecting relevant statistics.

2.2 `RFChannel`

A user of `RFChannel` needs to define a collection of virtual methods describing the channel properties. The most important functions are listed here. (For the sake of brevity, the arguments and the return data types are not presented.):

- `RFC_att()` describes signal attenuation depending on the distance and possibly other attributes that can be extracted from the source and destination transceivers.
- `RFC_act()` is used to tell whether the transceiver senses any signal at all (carrier sense) based on the total combined signal level arriving at it at the moment. A parameter specifies the receiver sensitivity, which is one of settable parameters of a transceiver. The sole purpose of this method is to determine when the channel is perceived *busy* or *idle* (and when to trigger the corresponding events `ACTIVITY` and `SILENCE`).
- `RFC_bot()` is responsible of assessing whether the beginning of a packet arriving at a transceiver is recognizable as such, i.e., the packet stands a chance of being received. Its arguments are the transmission rate (at which the packet was transmitted), the received signal level (the Received Signal Strength Indication, RSSI) of the packet, the current setting of receiver sensitivity, and pointer to the special data structure for the *interference histogram* (section 2.2.1), which stores the history of interference suffered by the packet's preamble.
- `RFC_eot()` determines whether the end of a packet arriving at a transceiver may trigger a successful reception of the packet. The arguments are exactly as for `RFC_bot`, except that the interference histogram the method can access applies to the main body of the packet rather than the preamble.
- `RFC_erb()` returns the randomized number of error bits within a specified sequence length of bits received, for a given received signal level, receiver sensitivity, and interference level. Its role is to describe the distribution of bit errors as a function of the signal to interference (SIR) ratio. The method is optional and needed only if the protocol program calls `errors` or `error` which force the flagging of actual "hard" errors.
- `RFC_erd()` returns the randomized number of received bits (under the conditions described by four of its arguments that are the same as in `RFC_erb` preceding the occurrence of a user-defineable "special" configuration of bits described by an additional argument. Typically, that configuration (the timing of its occurrence described as a bit interval) is strongly related to the distribution of bit errors (thus, `RFC_erd` is closely correlated with `RFC_erb`). For example, the method may return the number of bits preceding the nearest occurrence of a run consisting of a given number of consecutive bit errors. The method is used solely for triggering `BERROR` events, whose interpretation is up to the protocol program. In particular, if the program is not interested in perceiving those events, the method need not be provided.
- `RFC_add()` is the centerpiece of calculating the aggregation of multiple signals arriving at the receiver at the same time. Contrary to its mnemonic name, the method does not sum the signals, but is free to combine them in whichever fashion best describes the physical medium. An argument specifies the number of signal level entries, which are in turn provided as an array of simple structures storing the signal levels of all individual activities perceivable by the transceiver. This operation is also used in calculating the total level of interference affecting one selected signal; in such a case, the signal in question is excluded from the calculation.

The default definitions of the above methods provide a complete but naive functionality. The default (functionally void) stubs for `RFC_erb` and `RFC_erd` raise errors when called, but it is possible to have a fully functional and non–trivial channel model that makes no reference to those methods. For example, one that retains "soft" signal values. Furthermore, the structure describing a signal level, `SLEntry` (section 2.3.1, includes a tag, which can be used to associate user–defined properties with the signals, e.g., codes for CDMA–type channels. Signal tags are accessible to, and can be interpreted by, `RFC_add` and can therefore affect the way the multiple signals are combined, e.g., taking into account more properties than the sheer signal strength at the receiver.

2.2.1 Interference Histogram

The interference histogram passed as the last argument to `RFC_bot` concerns solely the preamble component of the packet. On the other hand, the histogram passed to `RFC_eot` applies to the packet proper, excluding the preamble. In both cases, the histogram describes the complete interference history of the respective component as a list of different interference levels suffered by it within specific intervals.

The interference histogram is a class comprising several useful methods and two public attributes:

```
int NEntries;
IHEntry *History;
```

where `History` is an array of size `NEntries`. Each entry in the `History` array is a simple record that looks like this:

```
typedef struct {
    TIME Interval;
    double Value;
} IHEntry;
```

The sum of all `Intervals` in `History` is equal to total duration of the activity being assessed. The corresponding `Value` attribute of the `History` entry gives the interference level (as calculated by `RFC_add`) suffered by the activity within that interval. The complete `History` covers as many intervals as many different interference levels have happened during the perception of the assessed activity. If the preamble or packet has suffered absolutely no interference, its `History` consists of a single entry whose `Interval` spans its entire duration and whose `Value` is 0.0. Finally, additional methods are also provided to examine the interference history, as well as to extract the average, and the maximum values of the interference histogram over the entire time period, or of particular time fragments of the monitored activity.

2.3 `Transceiver`

The transmission functionality related to `Transceiver` is less interesting from the reception aspects, so we will focus exclusively on reception–oriented events for which a transceiver can wait:

- `BOT` (*Beginning of Transmission*) occurs as soon as the transceiver perceives the beginning of packet following a preamble, and `RFC_bot` returns `YES` for that packet. The interpretation of this event is that the preamble has been recognized by the transceiver, which can now begin to try to receive a packet.

- `EOT` (*End of Transmission*) occurs as soon as the following three conditions hold simultaneously: 1) the transceiver perceives the end of packet, 2) `RFC_bot` returned (earlier) `YES` for that packet, 3) `RFC_eot` returns `YES` for the packet in the current stage. The interpretation of this event is that a packet has been successfully received.

- `BMP` (*Beginning of My Packet*) occurs under the same conditions as `BOT` with one added necessary condition: the packet must be addressed to the station running the process that has issued the wait request. Formally, it means that a helper function `isMy()` returns `YES` for the packet (this also covers broadcast scenarios in which the current station is one of possibly multiple recipients).

- `EMP` (*End of My Packet*) differs from `EOT` in the same fashion that `BMP` differs from `BOT`, that is, the packet must be addressed to the station running the process that has issued the wait request. This is the most natural way of receiving packets at their proper destinations.

- `ANYEVENT` occurs whenever the transceiver perceives anything of potential merit (any change in the configuration of perceived activities), e.g., when any of the preceding events would be triggered, if it were awaited. Beginnings and ends of all packets not necessarily positively assessed by `RFC_bot`/`RFC_eot`, also trigger `ANYEVENT`. If any activity boundary or stage occurs at the moment when the wait request for `ANYEVENT` is issued, the event is triggered within the current time unit.

It is possible to directly monitor the level of interference suffered by a selected activity, typically a packet being in some partial stage of reception. The transceiver method `follow()` allows the program to declare the activity to be monitored (followed). If a non-NULL argument is present, it should point to a packet being carried by one of the activities currently perceived by the transceiver. Such a packet pointer can be obtained, e.g., via `ThePacket` by receiving a packet–related event `BOT` or `BMP`, or through explicit inquiries addressed to the transceiver about ongoing activity. If the argument is absent (or NULL), it implies the activity last examined by a transceiver inquiry. The method returns `OK`, if the argument (or the last transceiver inquiry) identifies an activity currently perceived by the transceiver (in which case the activity whose interference level is to be monitored has been successfully specified), or `ERROR` otherwise. Whether a packet will be `follow`ed or not is a matter of course left to the implementation. The point here is that a receiver may not "magically" know that a transmitter is sending something to it. It has to extract meaning out of the current level of channel activity and decide whether it is reasonable to expect that it could conceivably receive a packet. It may, for all practical purposes, decide to `follow` a packet only to identify, after the corresponding `EOT` that it was not really addressed to it. The example in section 3 illustrates one such case.

Finally, `Trasceivers` can wait on additional events available to them: `SILENCE`, and `ACTIVITY`, as defined earlier as well as `BOP`, and `EOP` which allow one to access the preamble activity (if necessary) independently of the assessment provided by `RFC_bot` and `RFC_eot`. Additional useful events are `SIGLOW`, `SIGHIGH`, `INTLOW`, and `INTHIGH` that identify when certain thresholds related to the received signal or to the interference are crossed.

2.3.1 Event Assessment

At any moment, a given transceiver may perceive a number of packets arriving from different neighbors and being at different stages. The role of the assessment procedure at a receiving transceiver is to determine whether any of those packets should be received or, more specifically, whether it should trigger the events that will conceptually amount to its reception. The decisions are arrived at by the collective interaction of the user-exchangeable *assessment methods*.

First, `RFC_att` determines the signal level of the packet at the perceiving port, based on transmission power and distance between the sender and the perceiving transceiver. These two arguments may be sufficient to determine the packet's received power—in those propagation models in which attenuation depends solely on distance, but `RFC_att` receives two additional arguments pointing to the source and destination transceivers. Based on their identity, the user is able to implement arbitrary attenuation criteria, including ones that have nothing to do with distance. For example, it could identify whether (currently) there is an obstacle between them.

The signal level returned by `RFC_att` is associated with the packet at its perceiving transceiver. Suppose that some transceiver v perceives n packets denoted $p_0, \ldots, p_{n-1}$ with $r_0, \ldots, r_{n-1}$ standing for their received signal levels. The interference level suffered by one of those packets, say p_k is determined by a combination of all signals of the remaining packets. Method `RFC_add` is responsible for calculating this combination. Generally, the method takes a collection of signal levels and returns their combined signal level. A single signal is described by the following structure:

```
typedef struct {
    double Level;
    Long Tag;
} SLEntry;
```

where `Level` is the received level of the signal as calculated by `RFC_att`, and `Tag` is the `Tag` attribute used by the transmitter when the packet was transmitted. The third argument of `RFC_add` is an array of such records; its size is determined by the first argument.

If the second argument of `RFC_add` is nonnegative, it is viewed as the index of one entry in the signal array that must be ignored. This is exactly what happens when the method is called to calculate the interference suffered by one packet. The collection of signals passed to the method in the third argument always covers the complete population of signals perceived by the transceiver. Thus, the indicated exception refers to the one signal for which the interference produced by the other signals is to be determined. Sometimes `RFC_add` is invoked to calculate the global signal level caused by all perceived packets with no exception. In such a case, the second argument is `NONE`, i.e., –1.

It may happen that the perceiving transceiver is transmitting a packet of its own at the moment when some external packets are being perceived. In such a case, the method should take into account the interference caused by the transmitter (in many realistic cases it will make any reception impossible). This is where the last argument of `RFC_add` comes into play. If not `NULL`, it points to a signal record (`SLEntry`) describing the transmission of the transceiver's own transmitter. If `NULL`, it means that the transmitter is silent. The `Level` attribute of that entry is just the transmission power level (`XPower`), and the `Tag` reflects the current setting of the transceiver's `Tag`.

The standard version of `RFC_add` built into `RFChannel` performs simple addition of all the signal levels and ignores the tags. This should be OK for simple wireless channels; however, some channels (notably CDMA) may want to diversify the impact of different signals on the total level of interference suffered by a packet, e.g., based on the code (which can be represented by the tag).

The actual decision regarding a packet's reception is made by `RFC_bot` and `RFC_eot`. Their meaning is similar, but they are called at different stages of the packet's perception. `RFC_bot` is invoked at the end of preamble and before the first bit of the actual packet. In physical terms, it determines whether the receiver has been able to recognize that a packet is arriving and, based on the quality of preamble, clock itself to the packet. If the decision is `NO`, the packet will not be received. In particular, its next stage (main body) will not be subjected to another assessment, and the packet will trigger no reception events. If the method returns `YES`, the packet will undergo another assessment, `RFC_eot`, which is invoked immediately after the last bit of the packet. Note that regardless of whether the earlier assessment by `RFC_bot` has been negative or positive, the packet will continue contributing its signal to the population of activities perceived by the transceiver until it is heard no more.

3 EXAMPLE: A SHADOWING CHANNEL MODEL

Let us have a look at a complete model of a shadowing channel. The primary objective of the model is to prescribe the probability that the arrival of a packet at a transceiver will result in its positive assessment at the stages of reception relevant from the viewpoint of the receiver program. To see what the low-level protocol program expects from the model, we shall start from the SMURPH code of the three processes implementing that protocol. For simplicity, we shall skip the description of the complete layouts of those processes, which, in addition to the code presented below, includes the *data area*, i.e., private and external attributes accessible to the process. We assume that the meaning and role of such attributes is obvious from the context. In particular, `IF` (for InterFace) stands for a pointer to the transceiver object, which formally belongs to the station running the

process. That station, in turn, is represented by the standard attribute S.

3.1 The Protocol

The transmitter process is as follows:

```
Xmitter::perform {
  state XM_LOOP:
    if (!S->ready (MinPl, MaxPl, Frame)) {
      Client->wait (ARRIVAL, XM_LOOP);
      sleep;
    }
    if (S->Receiving) {
      Rcv->wait (SIGNAL, XM_LOOP);
      sleep;
    }
    RSSI->signal (START);
    Timer->wait (LBTDelay, XM_LBS);
  state XM_LBS:
    RSSI->signal (STOP);
    if (RSSI->sigLevel () >= LBT_THRESHOLD) {
      Timer->wait (genBackoff (), XM_LOOP);
      sleep;
    }
    IF->transmit (Buffer, XM_DONE);
  state XM_DONE:
    IF->stop ();
    Buffer->release ();
    proceed XM_LOOP;
}
```

At the top of its main loop (state XM_LOOP), the process checks if there is a packet to transmit (method ready belonging to the process's owning station) and, if it is not the case (ready returns false), it idles awaiting a packet's arrival (the wait request issued to the Client). Having acquired a packet, the transmitter makes sure that the receiver process is not currently in the middle of a packet reception (flag Receiving), in which the process will avoid interfering until it receives a "go" signal from the other process. As a rudimentary technique of collision avoidance, before commencing the actual transmission, the transmitter wants to monitor the wireless channel for a certain amount of time (LBTDelay) to detect a possible ongoing activity in its neighborhood. This is accomplished with the assistance of an auxiliary process (of type ADC) pointed to by RSSI.

```
ADC::perform {
  double DT, NA;
  state ADC_WAIT:
    this->wait (SIGNAL, ADC_RESUME);
  state ADC_RESUME:
    ATime = 0.0;
    Average = 0.0;
    Last = Time;
    CLevel = IF->sigLevel ();
    IF->wait (ANYEVENT, ADC_UPDATE);
    this->wait (SIGNAL, ADC_STOP);
  state ADC_UPDATE:
    DT = (double)(Time - Last);
    NA = ATime + DT;
    Average = ((Average * ATime) / NA) +
              (CLevel * DT) / NA;
    CLevel = IF->sigLevel ();
    Last = Time;
    ATime = NA;
    IF->wait (ANYEVENT, ADC_UPDATE);
    this->wait (SIGNAL, ADC_WAIT);
}
```

The process does nothing until it receives a signal (START from the transmitter), which will force it to state ADC_RESUME. There the process initializes variables for the calculation of average signal level and updates them in response to any change in the configuration of activities perceived by the transceiver (ANYEVENT). This procedure stops when the transmitter notifies the process that the monitoring period is over (the STOP signal).

At the end of the monitoring period, the transmitter calls the sigLevel method of the ADC process to return the calculated average signal level perceived by the transceiver. If the signal is above the threshold, the process concludes that the channel is busy and backs off for a randomized amount of time (prescribed by genBackoff). Otherwise, the channel is assumed idle, and the transmission begins. When completed (in state XM_DONE), the process terminates the transmission and marks the packet buffer as empty.

At this stage, the semantics of the channel model are manifest in the values returned by IF->sigLevel() in the ADC process. Those values reflect the combined signal level perceived by the transceiver at any instant of the monitoring interval, after accounting for the possibly multiple points of origin of its components, including their attenuation. The receiver process expects a bit more functionality from the channel model:

```
Receiver::perform {
  state RCV_WAIT:
    S->Receiving = NO;
    this->signal (GO);
    IF->wait (BOT, RCV_START);
  state RCV_START:
    S->Receiving = YES;
    IF->follow (ThePacket);
    skipto RCV_RECEIVE;
  state RCV_RECEIVE:
    IF->wait (EOT, RCV_GOTIT);
    IF->wait (BERROR, RCV_WAIT);
    IF->wait (BOT, RCV_START);
  state RCV_GOTIT:
    if (ThePacket->isMy ())
      Client->receive (ThePacket,
                       TheTransceiver);
    proceed RCV_WAIT;
```

```
}
```

As most of the complexity of the reception model is hidden in the implementation of the wireless channel, the receiver structure is deceptively simple. The process wakes up on the `BOT` event triggered on the transceiver. Note that this event captures the assessment of the first important reception stage, i.e., the moment when the transceiver recognizes a packet beginning. Following that event, in state `RC_START`, the process executes `follow`, to indicate that the packet's fate should now be traced, and moves to state `RCV_RECEIVE` with one time unit delay to ensure that the `BOT` event is no longer present on the transceiver. Then, in state `RCV_RECEIVE`, the receiver awaits the first of three possible outcomes: 1) an `EOT` event (which will mark the positively assessed reception of the packet's last bit), 2) `BERROR`, which will indicate a lost reception, and 3) another `BOT` event, meaning that another receivable packet, possibly arriving at a stronger signal than the first one, has taken over (normally, such an event should be preceded by a lost reception). Upon `EOT`, the process moves to state `RCV_GOTIT` where the packet is formally received, if it turns out to be addressed to the node running the process (`isMy` returns true).

3.2 The Channel

The channel model is declared this way:

```
rfchannel RFShadow {
  double NBeta, Sigma, LFac, BNoise, RDist,
    CDist;
  Long MinPr;
...
  private methods
  assessment method
  void setup (...)  ...
};
```

where the non-method attributes represent the parameters and are set by the standard `setup` method. According to the shadowing propagation model, signal attenuation is described by the following formula:

$$\frac{P_d}{P_{d_0}} = -10\beta log(d/d_0) + \sigma$$

where P_d is the power level of the received signal at distance d, P_{d_0} is the power level at some reference distance d_0, β is the loss exponent and σ is a lognormal (Gaussian) random component with a given standard deviation. To facilitate the calculation of P_d at the destination, `NBeta` is set to $-\beta$, `RDist` is set to d_0, and `LFac` is precomputed as $\texttt{RDist}^{\beta}/l_0$, where l_0 is the power loss at distance d_0, specifically, $l_0 = P_x/P_{d_0}$, where P_x is the transmit power at the source. Then, the following assessment method takes care of calculating the signal loss:

```
double RFC_att (double xp, double d,
       Transceiver *src, Transceiver *des) {
  return (d > RDist) ?
    xp * LFac *
    dBToLin (dRndGauss (0.0, Sigma)) *
    pow (d, NBeta) : xp;
};
```

where `Sigma` stands for the standard deviation of the σ component. Note that this component is log-normalized by `dBToLin`, which converts decibels to linear. This is because the model requires power levels and their ratios to be expressed linearly. Also, it is assumed that nodes never get closer than the reference distance `RDist`, which thus becomes the minimum separation distance.

The model does not redefine the default `RFC_add` method, which assumes that multiple signals at the receiver combine additively. Here we see how the first stage of packet reception is assessed:

```
Boolean RFC_bot (RATE r, double sl,
                 double sn, const IHist *h) {
  return h->bits (r) >= MinPr &&
         !error (r, sl, sn, h, -1, MinPr);
};
```

The interference histogram (type `IHist`), whose pointer is passed as the last argument to `RFC_bot`, stores the interference history of the packet as received so far. To be deemed receivable, a packet must be preceded by at least `MinPr` bits of preamble, and none of those bits must have been received in error. The argument list of `error` includes: the transmission rate `r` (needed to transform time intervals into bits and vice versa), the received signal level `sl`, the receiver sensitivty `sn` (ignored in out model), the interference histogram `h`, and the specification of the packet fragment (from-to expressed in bits) to be examined. The present arguments describe the last `MinPr` bits of the packet portion received so far, i.e., the last `MinPr` bits of the preamble.

The end-of-packet assessment is even simpler:

```
Boolean RFC_eot (RATE r, double sl,
                 double sn, const IHist *h) {
  return !error (r, sl, sn, h);
};
```

This method simply says that to be correctly received a packet must have no bit errors. Note that the interference histogram at this stage excludes the preamble, which must have been positively assessed by `RFC_bot` for the present assessment to take place at all. The actual fate of individual bits is determined by this method:

```
Long RFC_erb (RATE tr, double sl, double rs,
              double ir, Long nb) {
  return lRndBinomial( ber(sl/(ir + BNoise)),
                           nb);
};
```

which returns the randomized number of bits within a run of `nb` bits received at the signal level `sl` and interference level `ir`. Argument `rs`, representing the receiver sensitivity, is ignored in our model (all receivers operate at the same fixed sensitivity level).

Note that method `ber` is one component of the model that must be supplied by the user. Its simple role is to transform the signal-to-noise ratio (*SNR*) into the probability that a single bit is received in error. The important feature of the assessment strategy is that the actual error rate applied to a packet component depends on the *SNR* of this particular component, and may differ depending on the varying level of interference during the packet's reception. For example, the standard `error` method referenced in `RFC_bot` and `RFC_eot`, which is applicable to arbitrary chunks of received bits, properly accounts for the fact that different runs of those bits may have suffered different interference levels (according to the interference histogram) and thus have been subjected to different error rates. The noise level used as the denominator in the argument to `ber` includes a fixed background noise component, which is also a parameter of the model.

The role of the `BERROR` event (intercepted by the transmitter process in state `RECEIVE`) is to represent custom conditions occurring at randomized intervals during a packet reception and possibly depending on the momentary bit error rate. A typical example of such a condition is the loss of synchronization to the received packet, which will abort the reception. For example, in balanced encoding, whereby four-bit (logical) nibbles are represented by six-bit (physical) symbols, the reception of an illegal symbol amounts to such a scenario. Given a (dynamic) bit error rate, it is the role of this method to calculate such randomized intervals:

```
Long RFC_erd (RATE tr, double sl, double rs,
              double ir, Long ver) {
  double er = ber (sl/(ir + BNoise));
  return (er = dRndPoisson (1.0/(er*er))) >
                  (double) MAX_Long ?
                  MAX_long : (Long) er;
};
```

In this intentionally simplified case, the method returns a randomized interval (expressed in bits) until the first occurrence of a double-bit error, i.e., two consecutive incorrectly received bits. Similar to other assessment methods, it is correctly applied to chunks consisting of fragments received at different levels of interference (and thus different error rates). Once a process issues a wait request for `BERROR`, the system takes care of all this automatically by monitoring different levels of interference suffered by the received packet, and dynamically recalculating the interval until the awaited configuration of bits "occurs by chance." The last argument of `RFC_erd` (ignored in this case) can be used to select among different scenarios of interesting events.

4 CONCLUSIONS

We introduced extensions to SIDE/SMURPH with the specific intent to support the accurate simulation of wireless channels, and wireless networks with potentially large number of nodes. The abstractions introduced were tailored to express advanced wireless channel models, within reasonable overhead and user effort demands. An advanced wireless channel model cannot be simply a bit pipe with random independent bit errors. Therefore, we pushed the level of detail to a level where it is possible to capture and examine the variations of signals as they are perceived at transceiver endpoints. The signal strength of several combined transmissions, and the interference of such combined transmissions against a specific signal, are exposed to the simulation programmer and allow one to plug-in realistic channel models describing the space– and time–variable aspects of the channel.

REFERENCES

Kotz, K., C. Newport, R. S. Gray, J. Liu, Y. Yuan, and C. Elliott. 2004. Experimental evaluation of wireless simulation assumptions. In *Proceedings of the 7th ACM international symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2004)*, 78–82.

Haq, F. and T. Kunz. 2005. Simulation vs. emulation: Evaluating mobile ad hoc network routing protocols. In *Proceedings of the International Workshop on Wireless Ad-hoc Networks (IWWAN 2005)*, London, England, May 2005. Available online via <http://www.ctr.kcl.ac.uk/iwwan2005/papers/56.pdf> [accessed June 5th, 2006].

Gburzynski, P. 1996. *Protocol design for local and metropolitan area networks*. New Jersey: Prentice Hall.

AUTHOR BIOGRAPHIES

PAWEL GBURZYNSKI is a professor at the Computing Science Department of the University of Alberta. He received his M.Sc. and Ph.D. from the University of Warsaw in 1976 and 1982 respectively. He is a member of the editorial board for the International Journal of Communication Systems (Wiley). His e-mail address is <pawel@cs.ualberta.ca>.

IOANIS NIKOLAIDIS is an associate professor at the Computing Science Department of the University of Alberta. He received his M.Sc. and Ph.D. from Georgia Teach in 1991 and 1994 respectively. He is a member of the editorial board for Computer Networks (Elsevier) and for the IEEE Network Magazine. His e-mail address is <yannis@cs.ualberta.ca>.