

Model-based Fault Detection in Powerline Networking *

Anish Arora[†]
Ohio State University
Columbus, OH

Rajesh Jagannathan
Ohio State University
Columbus, OH

Yi-Min Wang[‡]
Microsoft Research
Redmond, WA

Abstract

Powerline networking is increasingly becoming an important component of home networking systems. Its reliability is however still a problem. In this paper, we consider the popular X10 powerline communication protocol. Our main contribution is a model-based fault detection system that achieves completeness of coverage for X10 faults. We develop experimentally a finite state automaton that models all legal sequences of X10 commands. The task of detecting every violation of this model is complicated by the presence of hidden state and unobservable illegal transitions. This problem is addressed by deducing the model state indirectly from the sequence of X10 commands that is observed on the powerline. To this end, we formulate the model state deduction task in terms of the observability of the model, a concept which arises in discrete-event dynamic systems. Based on the observability property of our X10 model, the detection of model violations is performed in our current implementation via regular expressions on observable X10 command sequences. Our final contribution is a preliminary diagnosis procedure for identifying X10 faults.

Keywords: formal methods, models, finite state automata, fault detection, hidden state, unobservable transitions, observability, network protocols, X10.

*This work was partially sponsored by DARPA contract OSU-RF #F33615-01-C-1901, NSF grant NSF-CCR-9972368, an Ameritech Faculty Fellowship, and a grant from Microsoft Research.

[†]Tel:+1-614-292-1836; Email:{anish}@cis.ohio-state.edu; Web:<http://www.cis.ohio-state.edu/~anish>

[‡]Email:ymwang@microsoft.com; Web:<http://research.microsoft.com/~ymwang>

1 Introduction

The powerline is the most commonly available wired connectivity in a majority of homes. Powerline networking therefore forms an important component of home networking systems, including Aladdin [1, 2] which is aimed at building a dependable and extensible home networking system and uses the X10 [3] powerline communication protocol. X10 enables X10-compatible devices, which are electrical components directly plugged into wall outlets, to communicate with each other. But these devices are susceptible to damage by voltage spikes. Additionally, signal attenuation and line noises generated by household appliances or external sources can transiently interfere with X10 communications. With neighboring houses sharing the same powerline subnet, X10 commands from one house can interfere with devices in another house. As a result, reliability remains as a major issue in X10 powerline networking.

Since faults—including, as we learned in our experience with Aladdin, complex and unanticipated ones—are unavoidable in X10 networking, we are motivated to detect their occurrence, and, if possible, to recover from them. Faults manifest themselves as anomalous behavior on the powerline in terms of illegal sequences of X10 commands. In this paper, we describe a general and extensible model-based fault detection system that can be deployed as part of any powerline networking application running X10 protocol.

We use a finite state automaton to model the legal sequence of X10 commands that can be generated on the powerline by the various X10 devices. Any behavior observed on the powerline which deviates from the model indicates a fault. The task of developing the model is however complicated by the fact that the X10 protocol is under-specified with respect to when exactly modules get to be addressed and unaddressed. Only after experimentation with various command sequences could we formulate the rules that govern the addressing of X10 modules and develop a model for the legal X10 command sequences.

Our detection system therefore involves monitoring the powerline communications and continuously detecting whether they imply illegal transitions in the model. This monitoring problem would be easily solved if all of the state and the transitions were visible to the detection system. Unfortunately, neither of these assumptions holds in the case of X10. For one, the state of the various individual X10 modules with respect to addressing cannot be directly observed. In other words, the state of model is hidden from the detector and thus even initializing the model is a problem. And two, not all illegal behaviors result in observable powerline communications; in other words, not all illegal transitions in the model can be directly observed.

Our detection system deals with the unobservability of state and illegal transitions by deducing the model state indirectly from the sequence of commands observed so far. The property of being able to determine the state based on only the transitions of the system is termed state observability. We prove the observability property of our model, by exploiting a construction in [4]. We then reduce the task of deducing the hidden state to that of observing specified regular expressions in the sequence of X10 commands. The net result is that if our detector is initialized correctly to the model state and no unobservable transitions occurs, it *always* detects every observable illegal transitions correctly. If, however, the detector is not properly initialized or (any finite number) of unobservable transitions occur, then the detector *eventually* deduces the correct state, detects the occurrence of some unobservable transition(s), and subsequently detects all observable illegal transitions correctly. It is in this sense that our detector “continuously” detects all illegal behavior.

An additional feature of our detector implementation is that it is self-stabilizing. In other words, it is self-tolerant to the corruption of its own state (as opposed to the state of the model that it tracks). Specifically, starting from an arbitrary detector state, the detector eventually reaches a

state from where it continuously detects all illegal behavior.

Contributions. As far as we know, ours is the first attempt to obtain a complete model for the valid command sequences for X10 protocol. Our current detector implementation continuously detects all behavior that is illegal with respect to this model; as noted above, our implementation deals with unobservable state, with incorrect initialization, with the occurrence of any finite number of unobservable transitions, and with the transient corruption of the state of the detector itself. Moreover, to support the task of fault diagnosis, we group illegal transitions together to form base patterns, and outline a preliminary procedure for diagnosis, along with some illustrative examples of faults observed in Aladdin and faults one can expect to observe.

Related Work. Other approaches to fault detection include characterizing the illegal sequences generated by a fault as a pattern [5] or modeling illegal behavior in terms of expert system rules [6]. Detection of a pattern or triggering of a rule, as the case may be, indicates the occurrence of the corresponding fault. Such approaches to fault detection are not always easily extensible, since a new fault not captured by the patterns or rules remains undetected. In contrast, the model-based approach achieves completeness of coverage, in that any new fault that violates the model is still detected.

Finite state automata have been used previously to model communication protocols for purposes of fault detection [7, 8, 9]. In these works, the system being modeled is a single agent that participates in the communication, whereas we model solely the communication medium as a single logical entity. Along similar lines, finite state automata have been used in the context of intrusion detection, alongwith other formal models. Ko [10] has proposed the use of formal specifications for anomaly detection, where an anomaly essentially denotes any possible violation of legal behavior, and Büschkes [11] has refined this approach to transaction-based specifications of the legal behaviors of communication protocols such as TCP. The complications of unobservable state and transitions and faults in the detection process itself are not dealt with in any of these approaches.

Unobservable states and transitions are dealt with in the literature on discrete event systems. A variation on the observability property is used in the construction of a detector (termed diagnoser) in [12]. In their model, faults are modeled not as illegal transitions but as illegal system states. The model state is estimated based on partially observed state outputs. The implementation of their detector involves simulation of an observer automaton. By way of contrast, our reduction of the detection to regular expressions obviates the need for explicit simulation of automata. A notion related to observability, namely diagnosability, is proposed in [13]. Diagnosability requires the detection of an illegal transition based on observed system transitions, but only after the fact and within a finite number of system transitions. However, diagnosability does not require all illegal transitions be detected. We, on the other hand, are interested in detecting all occurrences of illegal transitions in the sense described above.

Outline of the Paper. In Section 2, we discuss the specifics of X10 protocol addressing that we established through experimentation. In Section 3, we develop the model for legal X10 behavior. In Section 4, we discuss the implementation of the detector of model violations. In Section 5, we discuss some observed and other potential X10 faults. Finally, we present concluding remarks and discuss future work in Section 6.

2 X10 Protocol

In this section, we present the addressing logic obtained by experimentation, and the classification of functions that serves as the basis for the model. We start with the basics of X10 control. A typical system consists of multiple X10 modules attached to the common powerline communication medium. A CM11A PC interface attached to the PC via a serial port can be used to generate

and receive X10 commands on the powerline. Other modules act as receivers and control the household appliance attached to them. A two-way receiver can also respond to commands. An RF transceiver module converts RF signals from remote controllers into X10 commands. A sample powerline network is provided in Figure 1.

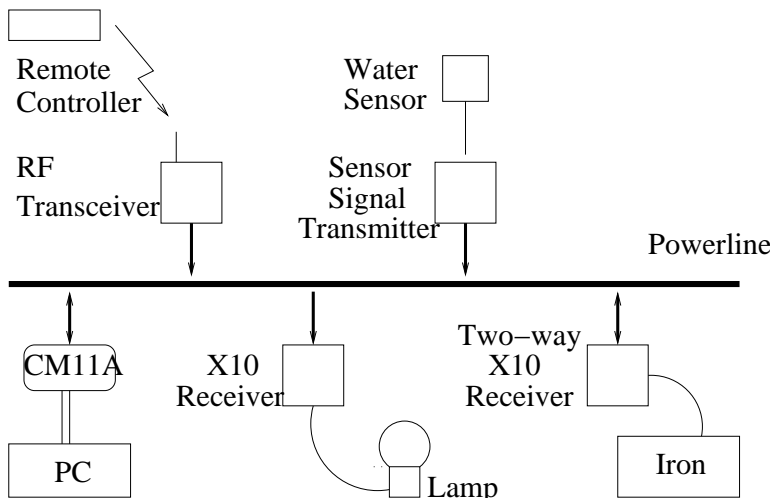


Figure 1: Typical Powerline Network

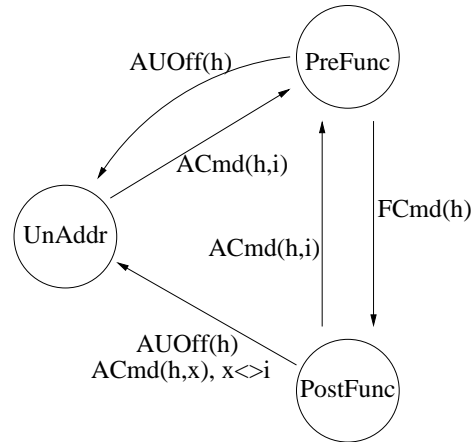
While the PC CM-11A interfaces are not assigned addresses, each X10 module is identified by an address which consists of a house code ($A \dots P$) and a unit code ($1 \dots 16$), which gives us a total of 256 distinct addresses. Each of the different types of modules react to some subset of sixteen function codes. The modules are usually controlled by issuing an address command consisting of a house code and unit code (e.g. 'A 1') which places the respective module in an addressed state, followed by a function command with the same house code and one of the sixteen function codes (e.g. 'A On'). The addressed module then responds to the function code specified as part of the function command.

2.1 Characterizing X10 Addressing Logic

The X10 protocol is under-specified as to when the modules move from unaddressed state to an addressed state and vice versa. Does an addressed module move to an unaddressed state immediately after executing the function command as described in the basic X10 control above? In order to figure out all the legal sequences of address commands and function commands that can be issued, we need to determine exactly when a particular module becomes addressed and unaddressed. We generated and experimented with various command sequences to ascertain the exact rules that govern the addressing logic in X10 control.

- **Addressing:** An address command consisting of house code h and unit code i addresses the module $[h, i]$.
- **UnAddressing:** A module can be unaddressed in two ways. The command $AUOff(h)$ unaddresses all modules with house code h . An address command with house code h and unit code i following a function command of the same house code h unaddresses all previously addressed modules with house code h (leaving the module $[h, i]$ addressed). (The reader is referred to Appendix D which summarizes the notation used in this paper.)

The state machine in Figure 2 captures the addressing logic for a module $[h, i]$. (The reader is referred to Appendix D for the notation used in this paper.) All transitions are labeled with the type of the command seen on the powerline by the module. All other commands, including the commands for a different house code, leave the module $[h, i]$ in the same state.



| State | Description |
|----------|---|
| UnAddr | $[h, i]$ unaddressed. |
| PreFunc | $[h, i]$ addressed, 0 function commands (house code h) issued. |
| PostFunc | $[h, i]$ addressed, 1 or more function commands (house code h) issued. |

Figure 2: Addressing Logic for $[h, i]$

2.2 Classifying X10 Functions

We classify the function codes into three classes — broadcast, unicast and multicast — based on the addressing logic. The function command `AllUnitsOff` was discussed as part of the addressing logic. The classification extends to the remaining function commands which contain the corresponding function codes.

- **Broadcast:** Modules respond to broadcast commands in both addressed and unaddressed states. Function commands `AllLightsOn`, `AllLightsOff` are classified as broadcast commands ($Brd(h)$).
- **Multicast:** Modules execute functions in this class only if they are in an addressed state. The multicast commands may be issued only when one or more modules have already been addressed. The functions `On`, `Off`, `Dim`, `Bright`, `PresetDim`, `ExtendedCode`, `ExtendedDataTransfer` make up this class. ($Mul(h)$).
- **Unicast:** When the function command `StatusReq` is issued an addressed module responds with its status - `StatusOn` or `StatusOff`. If more than one module has been previously addressed the responses generated by the individual modules cannot be distinguished. Issuing the function command `StatusReq` is valid only when exactly one module has been addressed. Therefore the command `StatusReq` along with the responses `StatusOn` and `StatusOff` are classified as unicast commands. ($Uni(h)$).

Next, we develop the model of valid X10 command sequences given the addressing logic and function classification discussed above.

3 Modeling Legal X10 Sequences

Logically, X10 is a simple communication protocol where messages do not contain the identity of the transmitter (PC interface, transceiver module) or the receiver module. Given that the communication medium is a broadcast medium, it is not possible to identify the sequence of commands sent by any single transmitter alone. Moreover, the X10 modules attached to the powerline usually tend to be 'dumb' devices and hence it is not feasible to monitor each and every device to detect if any fault has occurred. Therefore, we elected to model the entire powerline network (PC interfaces, modules and the medium). If we consider the set of X10 commands as an input alphabet, we can model the network as a finite state automaton generating X10 command sequences. The set of legal sequences is the language accepted by the model automaton.

One aspect of the X10 protocol that we exploit is that the commands with house code, say, h do not affect the addressing or functioning of modules with a house code different from h . In determining the set of legal sequences, the relative interleavings of the commands of different house code is not important, only the sequence of commands of the same house code. The model automaton is decomposed into a product of sixteen independent automata, one for each of the house codes h .

In the rest of this section, we develop the model automaton for a given house code h . First, we present the ideal model that does not take into account the common case faults. We then present the common-case faults and briefly describe how they are explicitly handled. Finally, we present the complete model that defines the system behavior completely.

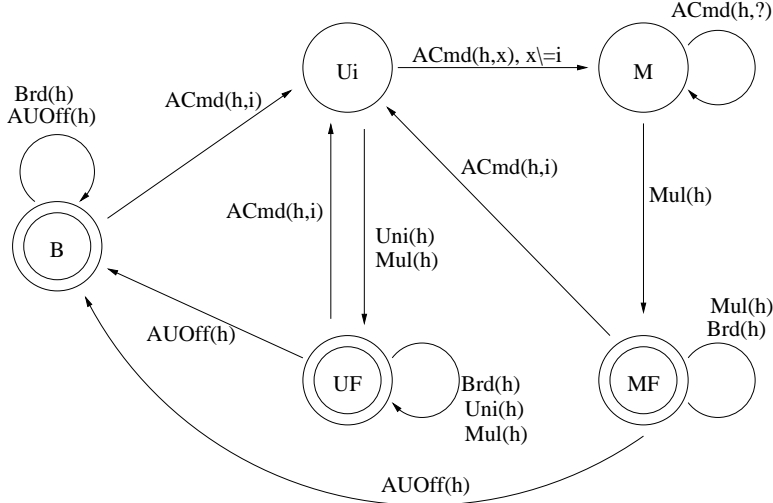
3.1 Ideal Model

The ideal model for the house code h is obtained by taking the product of the automata one each for $[h, i], i = 1, \dots, 16$, (Figure 2) and simplifying it by collapsing similar states. The simplified automaton of the ideal model for house code h is provided in Figure 3. The state B corresponds to the state where all the modules $[h, i]$ are in *UnAddr* state. The state U_i , parameterized on i , corresponds to state when $[h, i]$ is in *PreFunc* and the other modules are in the state *UnAddr*. The rest of the states simply collapse into the states UF, M, MF . The example in Figure 7 illustrates the difference between the states traversed in the individual automata for modules versus the states traversed in the ideal model.

The transitions are labeled with the X10 commands observed/generated on the powerline. Based on the classification of the functions, the distinction is made between states where only one module is addressed (states U_i, UF) and states where more than one module has been addresses (states M, MF). In states U_i, UF we can execute unicast functions and multicast functions, while in the states M, MF where more than one module has been addresses only multicast commands are executed. In state B only the broadcast commands are to be issued. The ideal initial states are indicated by a double-circle.

3.2 Complete Model

Although the ideal model specifies the legal sequence of X10 commands, it is not complete with respect to the states reached due to the occurrence of an illegal transition. This information is



| | | | |
|----|---|----|---|
| B | 0 modules addressed | | |
| Ui | 1 module $[h, i]$ addressed, 0 functions issued. | M | 1 or more modules addressed, 0 functions issued. |
| UF | 1 module addressed, 1 or more functions issued. | MF | 1 or more modules addressed, 1 or more functions issued. |

Figure 3: Ideal Model (house code h)

necessary for continued monitoring and detection of the system following the occurrence of an illegal transition. The automata in Figure 4 specifies the complete behavior of the system including both legal and illegal transitions (cf. the notations adopted in Figure 3). The transitions denoted by solid lines indicate legal transitions allowed by the model and dashed lines indicate illegal transitions.

The valid transitions in the complete model include, as expected, all the valid transitions in the ideal model. However, the ideal model is too restrictive and, if used as is, would result in a large number of unnecessary detections. We therefore expand the ideal model to account for (some, if not all, of) the common case faults. We assume the loss of a single command, for example due to signal attenuation or collision, to be a common-case fault. The assumption here is that the underlying system can handle the common case faults. Consider the loss of the function command from the sequence $A1 A0n$ followed by the retransmission of the same sequence. This results in the sequence $A1, A1, A0n$ being seen on the powerline which is valid with respect to the complete model while not allowed by the restrictive ideal one.

The majority of the transitions in the complete model which are carried over from the ideal model require little explanation. The other transitions are discussed below.

- *Invalid Addresses:* In states UF and MF any address command, valid ($ACmd(h)$) or otherwise ($Inv(h)$) causes previously addressed modules to be unaddressed. However, in the case of $Inv(h)$ since no existing module can be addressed the state reached is B (instead of Ui).
- *Maximum Repeats:* Even though the transitions on $ACmd(h)$ (and $Inv(h)$) in the states Ui and M are self loops, the model does not allow address commands to be repeated more than MUR times. Although we could have explicitly modeled the repeats using automaton states,

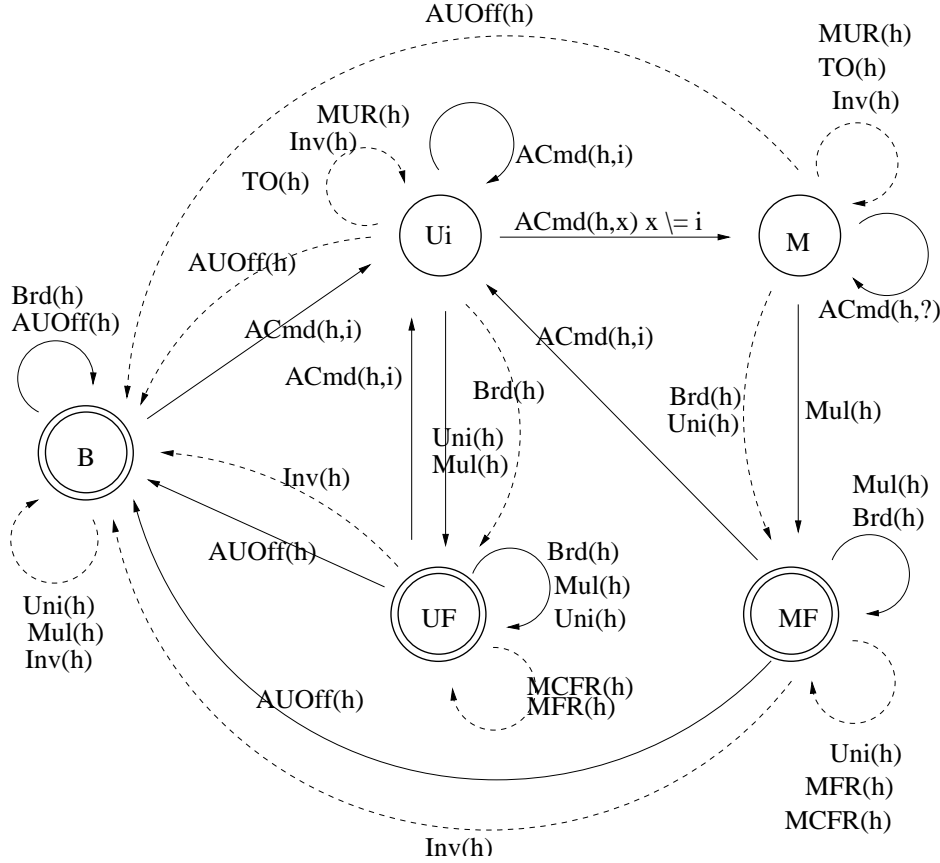


Figure 4: Complete Model (house code h)

we chose not to do so to keep the model simpler. Similarly, the parameter $MCFR$ limits the maximum number of continuous function commands with the same function code in states B, UF and MF . The transitions in the model labeled $MUR(h)$, $MFR(h)$ and $MCFR(h)$ indicate the occurrence of more than the allowed repetitions.

- *Timeouts:* The states B, UF, MF are reached after a logically complete command sequence has been issued (e.g. a broadcast command, or an address command followed by corresponding function commands). When in states Ui and M we expect the rest of the sequence to be issued and progress made. A timer is started whenever the states Ui, M are entered. The lack of progress in these states is captured by the expiration of the timer - denoted by the event $TO(h)$.

Since the complete model is the model of interest we shall henceforth refer to it simply as the 'model', unless stated otherwise. All legal sequences traverse only the legal (solid) transitions in the model. An illegal behavior therefore traverses at least one illegal transition. To detect illegal behavior we therefore detect illegal transitions (dashed) of the model automaton, which is discussed in the next section.

4 Detection of Illegal Transitions

Our detector is the component whose task is to monitor the powerline and detect the occurrence of invalid transitions. Recall that the detection of illegal transitions is complicated in X10 due to the presence of *hidden* state and *unobservable* transitions. Without these complications, the detector would simply simulate the model automaton synchronously with the system transitions and detect the invalid transitions as and when they occur. In this section, we describe an implementation of the detector which does not require explicit initialization and handles a finite number of occurrences of unobservable transitions.

X10 does not have any provision for polling the addressing state of the modules themselves. (The `StatusReq` command described in the previous section is only useful for determining the On/Off state of two-way X10 receivers.) The state of the model is therefore *hidden* and cannot be observed directly by the detector. To motivate unobservable transitions, let us provide an illustrative example: Consider that the power to a room is controlled via a master switch, and assume the user inadvertently turns the master switch off. All modules physically in the room and attached to the powerline are reset and lose any prior addressing information. Once power is restored modules restart in an unaddressed state. The state of system in this case was changed arbitrarily without any command being issued on the powerline and is modeled as an unobservable transition. (Also, as noted before, transient faults that can also arbitrarily corrupt the state of the detector may also be modeled as unobservable transitions.)

4.1 Observability of X10

Although the model state cannot be observed directly, our detector can observe the X10 protocol commands generated on the powerline. The state of the model can be deduced indirectly from the observable sequence of transitions. For example, if the observed sequence of commands is *A On, A 1*, assuming that no unobservable transitions occur in the interim, then irrespective of what the state of the model is initially, its final state will be *U1*. This information can be used to synchronize the detector state with the model state. We make the assumption that the unobservable transitions occur only finitely. This reduces the problem of handling unobservable transitions to being able to predict the state of the system and resynchronize once the transitions have stopped occurring. Given the above formulation, the problem of initializing the detector at start-up also reduces to synchronization of states.

The problem of deducing the state of the model based solely on the observable transitions is a well studied problem in the area of discrete-event dynamic systems. Given any labeled transition system $G = (Q, \Sigma, \delta)$, where Q is a set of states, Σ is a set of labels, and δ is a set of transitions, *observability* is the property of being able to uniquely predict the system state at intermittent points [4]. Intuitively, G is observable if the ambiguity between any two states p, q is removed in a bounded number of transitions for all possible behaviors starting from p and q . At that point, the system state is uniquely known and the detector state can be synchronized with it. Observability of the system can be checked [4] by constructing the product automaton $P = (Q \times Q, \Sigma, \delta_P)$ where δ_p is defined such that

$$\delta_p((p, q), \sigma) = (\delta(p, \sigma) \cup \delta(q, \sigma)) \times (\delta(p, \sigma) \cup \delta(q, \sigma))$$

The set of unambiguous states in P is $E = \{(p, p) | p \in Q\}$. The necessary and sufficient condition for G to be observable is that all cycles in P pass through a state in E .

This condition is true for our X10 model and so the model is observable. For X10, $Q = \{B, Ui, UF, M, MF\}$, Σ is the set of X10 commands and events, and δ consists of all solid and dashed

transition in Figure 4. In Appendix B, we give the product automaton and a sketch of proof that this condition is true in our system. (The complete proof is omitted for reasons of space.) Since G is deterministic, once one of the states is uniquely determined, the detector possesses perfect knowledge of the states that follow. It follows that even when unobservable transitions occur, the detector will eventually know the correct state, deduce that some unobservable transition has occurred in the interim, and subsequently track the model correctly.

4.2 Regular-Expression-Based Detector Implementation

The observability property of our X10 model also implies that for each state there exists a sequence of transitions that uniquely determine the state irrespective of the start state. (Again, a complete proof is relegated to the full version of this paper.) The table in Figure 5 lists, for each system state, the sequence that allows the detector to uniquely determine that state at the end of the sequence. For example, after the occurrence of sequence ending in $Auoff(h)$ the system state is B and the observer automaton state is the singleton $\{B\}$. Every illegal transition can be identified by a state and a command that causes the illegal transition, for example, $\delta(M, Uni(h))$. By replacing the state with its corresponding sequence in Figure 5, every illegal transition can be expressed in terms of its sequence only. For example, $\delta(M, Uni(h))$ can be expressed as $ACmd(h, x)ACmd(h, y)ACmd(h, ?)^*Uni(h)$, $x \neq y$.

| State | Sequences |
|-------|---|
| B | $Auoff(h)$ |
| Ui | $Auoff(h) Acmd(h, i)^+, Fcmd(h) Acmd(h, i)^+$ |
| UF | $Auoff(h) Acmd(h, i)^+ Fcmd(h)^+, Fcmd(h) Acmd(h, i)^+ Fcmd(h)^+$ |
| M | $Acmd(h, x) Acmd(h, y) Acmd(h, ?)^* \}$ with $x \neq y$ |
| MF | $Acmd(h, x) Acmd(h, y) Acmd(h, ?)^* Fcmd(h)^+$ with $x \neq y$ |

Figure 5: Sequences for Observability

Currently, in Aladdin [1], the regular expression-based detector is implemented using C++ as a Windows DLL (dynamically linked library). The detector consists of two main components: a regular expression parser and an online regular expression evaluator. On initialization, the detector is supplied with a file specifying the regular expressions that are to be detected. A sample file is provided in Appendix C. The regular expression parser creates the requisite data structures. The monitoring component of the DLL is supplied with the X10 commands and events which are then used to evaluate the individual regular expressions. Multiple partial matches are maintained after processing each command. Much of the code for the DLL is reused from the implementation of the path expression matching in the SIEFAST [14] simulator. Our implementation runs on a PC, which receives the sequence of X10 commands on the powerline via a CM11A PC interface.

5 Identification of X10 Faults

The task of identifying the exact faults which yield model violations falls under diagnosis. While a complete discussion of automatic diagnosis of faults is outside the scope of this paper, we discuss in this section some X10 failures that we observed in the Aladdin network and others that are likely to occur. In each case, we identify the illegal behavior observed.

5.1 Base Patterns

To support the task of identifying faults, we group similar illegal transitions together and term a continuous occurrence (to avoid redundant detections) of such like illegal transitions together as a base pattern. Each base pattern is expressed as a regular expression on sequences of X10 commands and events alone. The table in Figure 6 provides the list of base patterns, the illegal transitions corresponding to each pattern and the regular expression form of each pattern. These base patterns are with respect to any given house code (for convenience, the base pattern without an accompanying house code parameter refers to the corresponding base pattern detected by at least one of the detectors for the different house codes.)

| Base Pattern | Illegal Transitions | Regular Expression |
|-----------------|--|---|
| InvalidAddr(h) | $(B, Inv(h))$ $(Ui, Inv(h))$ $(UF, Inv(h))$ $(M, Inv(h))$ $(MF, Inv(h))$ | $Inv(h)^+$ |
| UnAddressed(h) | $(B, Uni(h))$ $(B, Mul(h))$ | $AUOff(h)(Brd(h))^*$ $(Mul(h) Uni(h))^+$ |
| UnicastError(h) | $(M, uni(h))$ $(MF, uni(h))$ | $ACmd(h, x)ACmd(h, y)ACmd(h, ?)^*$ $(Brd(h) Mul(h))^*Uni(h)^+$ |
| Timeout(h) | $(Ui, TO(h))$ $(M, TO(h))$ | $ACmd(h, ?)^+TO(h)^+$ |
| Incomplete(h) | $(Ui, AUOff(h))$ $(Ui, Brd(h))$ $(M, AUOff(h))$ $(M, Brd(h))$ | $ACmd(h, ?)^+(AUOff(h) Brd(h))$ |
| AddrRepeat(h) | $(Ui, MUR(h))$ $(M, MUR(h))$ | $(ACmd(h, i)ACmd(h, ?)^*)^{MUR+1}$ |
| FContRepeat(h) | $(B, MCFR(h))$ $(UF, MCFR(h))$ $(MF, MCFR(h))$ | $(FCmd(h, f))^{MCFR+1}$ |
| FuncRepeat(h) | $(UF, MFR(h))$ $(MF, MFR(h))$ | $FCmd(h, ?)^{MFR+1}$ |

“*” means 0 or more repeats; “+” means 1 or more repeats; “|” means choice

Figure 6: Base Patterns

The first pattern $InvalidAddr(h)$ captures the fact an invalid address has been issued. The patterns $UnAddressed(h)$ and $UnicastError(h)$ capture the mismatch between the number of modules addressed and the functions issued thereafter. The patterns $Timeout(h)$ captures the lack of progress and if in the interim some other transmitter were to start issuing another command sequence then the pattern $Incomplete(h)$ shall be triggered. The last three patterns capture the illegal repetition of address and function commands. The detection of illegal transitions now reduces to task of detecting the corresponding regular expressions.

5.2 Observed Faults

In the actual deployment of the Aladdin home networking system, we have observed several instances of the two types of failures described below [1]. The first type was physical failures of PC interfaces, which caused them to generate random, illegal X10 command sequences. The pattern of bad behavior seen on the network was the transmission of K or more identical function commands (same house code and function code). This type of failure is detected as the following base pattern with $MCFR < K$

Faulty PC Interface: $FContRepeat(h)$, for some h .

The second type involved a X10 RF transceiver module, which converts signals from the wireless remote controllers into X10 commands on the powerline. The transceiver modules are unable to distinguish between valid RF signals and noise that results from RF interference. The RF noise usually translates to an illegal command sequence on the powerline. The illegal behavior seen on the powerline was: There were L or more transmission bursts on the powerline in each of which either M or more address commands with the same unit code were transmitted consecutively or N or more identical function commands were transmitted consecutively. This pattern of bad behavior can be expressed as the following combination of base pattern (with $M > MUR$ and $N > MCFR$):

RF Interference: $(AddrRepeat + FContRepeat)^L$

5.3 Potential X10 Faults

The faults listed below are some of the faults we have been able to identify. In each case we indicate the base patterns that are detected as a result of the occurrence of the fault.

- **Security Intrusion:** The legitimate programs in the system have knowledge of all the valid addresses that are actually assigned to some X10 modules in the deployment. For example, in Aladdin [2], this information is maintained in the lookup service. Whereas a security intruder who is not privy to this information is likely to generate invalid address commands. The model violation resulting from a security intrusion is captured by the base pattern *InvalidAddr*.
- **Transmitter Interference:** The multiple X10 transmitters in the system utilize the shared powerline communication medium. The devices themselves are very simple and do not perform any advanced carrier sense before transmission of the X10 commands. Therefore we expect the command sequences generated by different transmitters to collide. The usual result of signal collisions is a complete loss of the packets. If the sequences overlap partially then modules end up observing garbled X10 command sequences. The collisions can be detected if they show up as a mismatch in the addressing and the function commands issued as reflected by the base patterns *UnAddressed*, *UnicastError*, *Incomplete*.
- **Software/Hardware Crash:** The states B , UF and MF are logical accepting states and the system can remain in these states indefinitely without any commands being issued, while the states Ui and M are not. Therefore we expect progress to be made within a reasonable amount of time whenever the system enters one of the states Ui or M (by progress we mean a function command issued). A software fault at the level of the controlling software or a hardware fault at the level of the PC/CM11A interface/modules could cause the components to fail and stop. In such a failure mode no progress would be made and this is captured by the *Timeout* base pattern.

The faults listed above are only representative and not intended to be exhaustive. It is however the case that any fault that causes the model to be violated can be detected in terms of one or more of the base patterns.

6 Conclusion and Future Work

In this paper, we experimentally developed a model of the legal command sequences of the X10 protocol and used model violations to detect the occurrence of faults. Our other contributions included the implementation of a detector for illegal commands sequences which does not require any explicit initialization of the model state as well as its own state, so as to be able to deal with the complications of unobservable state and transitions. Our detector thus implicitly has the desirable feature of being self-tolerant to transient state corruptions.

Our discussion of failure modes provided some insight into the base patterns that are triggered by the occurrence of the various faults. In general, the illegal behavior resulting from a fault will be more complex than triggering one base pattern. Since the set of base patterns covers the space of illegal behaviors, it is possible to detect all faults automatically in terms of (combinations of) base patterns. Identifying a particular fault may however require the use of a more sophisticated fault diagnosis engine, which we plan to investigate in future work.

Our approach to model-based detection is part of a larger research effort in model-based dependability in networked and distributed systems (cf. <http://www.cis.ohio-state.edu/~anish/group/papers.html>). The X10 case study has provided some conceptual underpinnings that should be helpful in applying the approach to deal more complex network protocols. For protocols which have been specified carefully a formal model may be developed analytically (as opposed to experimentally), viz. TCP. Future work in this direction will have to deal with a technical challenge that did not arise in the X10 case study: Not every state of a concrete protocol implementation may be assumed to correspond to some state in the formal model of the protocol. In this case, properties of the compiler that produced the concrete implementation will need to be exploited, perhaps along the lines demonstrated in [15]. Moreover, techniques for state-space reduction in the verification of protocol observability will need further study.

References

- [1] Y.-M. Wang, W. Russell, and A. Arora. A toolkit for building dependable and extensible home networking applications. In *Fourth USENIX Windows Systems Symposium USENIX-WIN'2000, Seattle, 2000*, Feb 2000.
- [2] Y.-M. Wang, W. Russell, A. Arora, J. Xu, and R. Jagannathan. Towards dependable home networking: An experience report. In *International Conference on Dependable Systems and Networks (DSN 2000)*. IEEE, July 2000.
- [3] *X10 communication protocol*. <http://www.x10.org>.
- [4] C.M. Ozveren and A.R. Willisky. Observability of discrete event dynamic systems. *IEEE Transactions on Automatic Control*, 35(7):797–806, Jul 1990.
- [5] S. Kumar and E.H. Spafford. A pattern matching model for misuse intrusion detection. In *National Computer Security Conference*, pages 11–21, Oct 1994.
- [6] R.O. Yudkin. On testing communication networks. *IEEE Journal on Selected Areas in Communications*, 6(5):805–812, Jun 1988.
- [7] C. Wang and M. Schwartz. Fault detection with multiple observers. *IEEE/ACM Transactions on Networking*, 1(1):48–55, Feb 1993.
- [8] G. Noubir, K. Vijayananda, and H.J. Nussbaumer. Signature-based method for run-time fault detection in communication protocols. *Computer Communications*, 21(5):405–421, May 1998.
- [9] A. Bouloutas, G. W. Hart, and M. Schwartz. Simple finite-state fault detectors for communication networks. *IEEE Transactions on Communications*, 40(3):477–279, Mar 1992.
- [10] C. Ko. *Execution Monitoring of Safety-Critical Programs in a Distributed System: A Specification-Based Approach*. PhD thesis, University of California, Davis, 1996.
- [11] R. Buschkes and M. Borning. Transaction-based anomaly detection. In *Workshop on Intrusion Detection and Network Monitoring, USENIX Networking*, 1999.
- [12] H.S. Zad, R.H. Kwong, and W.M. Wonham. Fault diagnosis in discrete-event systems: Framework and model reduction. In *Proceedings of the IEEE Conference on Decision and Control*, volume 4, pages 3769–3774. IEEE, 1998.
- [13] M. Sampath, R. Sengupta, S. Lafortune, K. Sinnamohideen, and D. Teneketzis. Diagnosability of discrete event systems. *IEEE Transactions on Automatic Control*, 40(9):1555–1575, Sep 1995.
- [14] *SIEFAST Users's Guide*. <http://www.cis.ohio-state.edu/siefast>.
- [15] S. Kulkarni A. Arora, M. Demirbas. Graybox stabilization. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 389–400, 2001.

A Example

The example below for the house code A illustrates the transitions in the automata for the addressing logic of the modules (Figure 2) and the model (Figure 3) for the following sequence of X10 commands:

$A1$, AOn , $A2$, $A3$, $ADim$, $AUOff(A)$, $A1$, AOn

| Cmd | State of module $[A, i]$ | State of model A | Comments |
|------------|--|--------------------------------------|--|
| | all in <i>UnAddr</i> | B | all modules $[A, i]$ unaddressed |
| A1 | $[A, 1]$ in <i>PreFunc</i> | U1 | $[A, 1]$ is addressed |
| AOn | $[A, 1]$ in <i>PostFunc</i> | UF | $[A, 1]$ still addressed, $[A, 1]$ executes On |
| A2 | $[A, 1]$ in <i>UnAddr</i> $[A, 2]$ in <i>PreFunc</i> | U2 | $[A, 1]$ unaddressed, $[A, 2]$ addressed |
| A3 | $[A, 2]$ in <i>PreFunc</i> $[A, 3]$ in <i>PreFunc</i> | M | $[A, 2]$ and $[A, 3]$ addressed |
| ADim | $[A, 2]$ in <i>PostFunc</i> $[A, 3]$ in <i>PostFunc</i> | MF | $[A, 2]$ and $[A, 3]$ still addressed both $[A, 2]$ and $[A, 3]$ execute Dim |
| AUOff(A) | all in <i>UnAddr</i> | B | all modules $[A, i]$ are unaddressed, all modules $[A, i]$ execute Off |
| A1 | $[A, 1]$ in <i>PreFunc</i> | U1 | $[A, 1]$ addressed |
| AOn | $[A, 1]$ in <i>PostFunc</i> | UF | $[A, 1]$ still addressed, $[A, 1]$ executes On command |

Figure 7: Example: Addressing Logic and Ideal Model

B Product Automaton

The product automaton P of the system G is provided in the Figures 10, 9 and 8 below. Some of the states have been repeated in the figures for clarity. The set of unambiguous states E are denoted with the double lines. The transitions among the unambiguous states that are the same as in G are left out for the sake of clarity. The necessary and sufficient condition for the observability of G is that all cycles in P pass through the some state in E .

Note that in P all cycles either pass through only states in E or are self-loops on the ambiguous states. The self looping transitions in Figure 9 labeled with $ACmd(h, i)$ or $Inv(h)$ are limited by MUR parameter . Therefore either the other transitions are taken or the model violation $MUR(h)$ is detected. Similarly in Figure 8 the model violation $MFR(h)$ is detected if the other transitions are not taken. In all valid behaviors starting from any state where the ambiguity exists between the model and system state we reach states in E in bounded number of steps, thus proving the observability of the G .

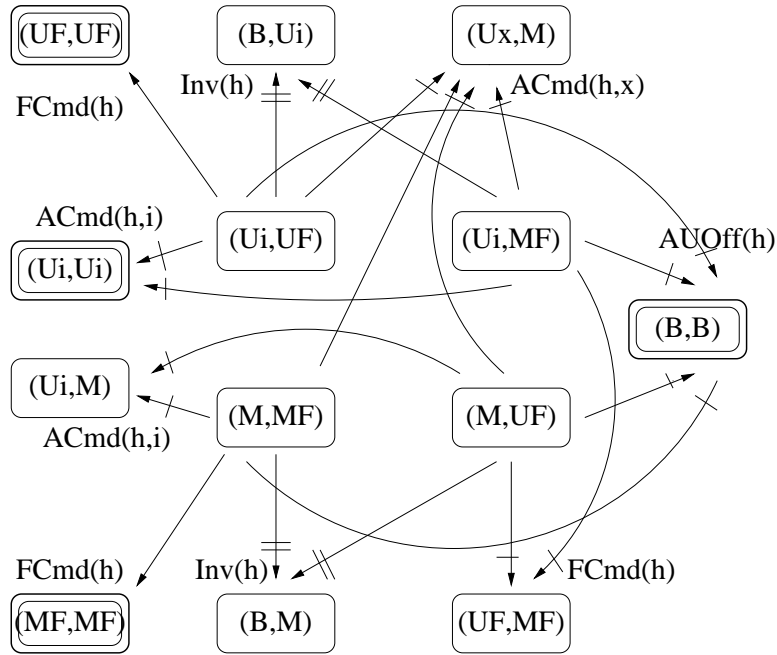


Figure 8: Product Automaton P (Part 1)

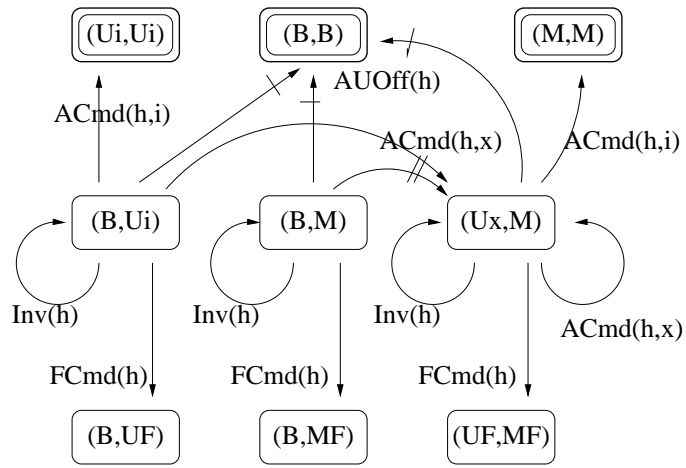


Figure 9: Product Automaton P (Part 2)

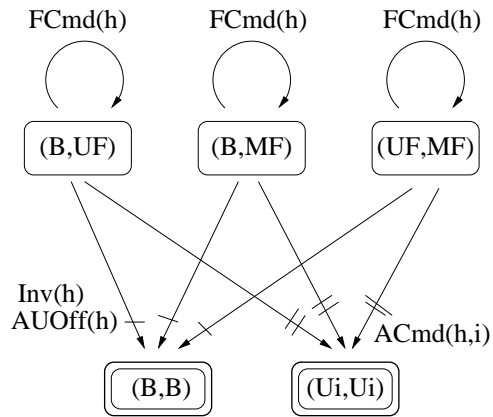


Figure 10: Product Automaton P (Part 3)

C Sample Source File

Below is a sample source file specifying the regular expressions that are to be detected.

parameters

u : 1 .. 16 ;

f : 0 .. 15 ;

expressions

AddrRepeat :

```
( ( { (IsAddress) and (HOUSE=='A') and (UNIT==u ) } )
+(
  { (IsAddress) and (HOUSE=='A') and (UNIT==u ) } ;
  { not (HOUSE == h) or (IsAddress) } *
)
)^4
```

FContRepeat :

```
( ( { not (IsAddress) and (HOUSE=='A') and (FUNC==f) } )
+(
  { not (IsAddress) and (HOUSE=='A') and (UNIT==f) } ;
  { not (HOUSE == h) } *
)
)^5
```

InvalidAddr :

```
( ( { ( IsAddress ) and (HOUSE == 'A') and not (IsValid ) } )
+( {true} * ; { ( IsAddress ) and (HOUSE == 'A') and not (IsValid ) } )
)
```

Figure 11: Sample file with regular expressions for house code A

D Notation

| | |
|----------------------|--|
| Parameters | |
| h | house code A, B, ... ,P |
| i, x | unit code 1, 2, ... 16 |
| f | function code On, Off, ... etc. |
| Module | |
| $[h, i]$ | X10 module with address (house code h , unit code i) |
| Address Cmds | |
| $ACmd(h, i)$ | valid address command (house code h , unit code i) |
| $ACmd(h)$ | valid address command (house code h , any unit code) |
| $Inv(h)$ | invalid address command (of a non-existent module) |
| Function Cmds | |
| $FCmd(h, f)$ | function command (house code h , function code f) |
| $FCmd(h)$ | function command (house code h , any function code) |
| $AUOff(h)$ | function command AllUnitsOff (house code h) |
| $Brd(h)$ | broadcast function command (house code h) |
| $Uni(h)$ | unicast function command (house code h) |
| $Mul(h)$ | multicast function command (house code h) |
| Events | |
| $TO(h)$ | timeout event in the detector |
| $MUR(h)$ | maximum number of contiguous address commands with the same unit code |
| $MFR(h)$ | maximum number of contiguous function commands |
| $MCFR(h)$ | maximum number of contiguous function commands with the same function code |

Figure 12: Notation