# Model-based design of dependability in distributed systems

Anish Arora, Rajesh Jagannathan, Yi-Min Wang

***Abstract*—**
**Distributed systems are notoriously subject to complex faults, some of which are unanticipated. Towards dealing with the problem of unanticipated faults, we describe in this paper a model-based approach to design of dependability. The model-based approach offers a potentially low-cost alternative to handling rare faults in a case-by-case manner, while allowing common faults to be handled individually. We illustrate the model-based approach with two case-studies: one concerning a home-network lookup service and the other an X10 powerline network.**

***Index Terms*—model, formal methods, dependability, unanticipated faults, concurrency, distribution, networking**

## I. Introduction

Distributed systems are notoriously subject to complex faults. A number of these faults are unanticipated, and one might argue that this number will only increase as distributed systems are increasingly built by dynamic composition of components from diverse sources (using technologies such as COM, CORBA, and JavaBeans). In spite of this trend, it is expected that the systems function acceptably in a variety of user and fault environments.

It is therefore our position that design of dependability for complex faults should also contend with unanticipated faults. Towards this end, we describe in this paper a model-based approach for dependability design in distributed systems.

### *A. Overview of model-based approach*

Notwithstanding the uncertainties associated with distributed system composition and their user/fault environments, their designer is typically in a position to specify the desired system behavior. The focal point of our approach is therefore a model of the desired behavior of the system, expressed in terms of a formal specification.

The effect of the faults on the system are only implicitly captured by considering potential violations of the model. By taking into account all possible model violations, the model-based approach contends with complex faults, both anticipated and unanticipated. It reduces the problem of dependability design to "enforcing" the system model at all times. Enforcement may occur at several levels: sometimes, it may be possible to ensure that the system behavior is never violated; at other times, the behavior may be recovered to (re)satisfy the system model; and occasionally the best that may be possible is to provide notification that the behavior has violated the model. (Details of how enforcement is achieved will be discussed to Section II.)

It should be stressed that model-based design is compatible with several extant approaches that are fault-centric. The compatibility lies in two senses: First, model-based design does not preclude fault-centric design of dependability to handle common-case faults, which are often simple and best handled explicitly. When it comes to rare-case faults, which tend to be more complex, interdependent, and difficult to diagnose, model-based design offers the possibility of dealing with them in an implicit manner. (The combined use of fault-centric design to handle simple, common-case faults and model-based design to handle complex, rare-case faults will be illustrated via a case study in Section III.) And second, model-based design does exploit knowledge of faults when designing the "enforcers" of the system model. (This will be illustrated in the case studies in Sections III and IV.)

Model-based design thus separates system dependability from system functionality. It allows the two to be modified independently. It also allows the dependability design to be reused for other systems which are expected to satisfy the same system model. Since it builds on more knowledge of the system than do approaches that treat the system as essentially a black-box, it offers the promise of lower-cost design of dependability. In particular, in the absence of faults, it allows the overhead of the dependability to be kept low, since essentially only the conformance to the model has to be checked. Moreover, in case the model evolves or

This work was partially sponsored by DARPA grant OSU-RF 01-C-1901, NSA grant MDA904-96-1-0111, NSF grant NSF-CCR-9972368, an Ameritech Faculty Fellowship and a grant from Microsoft Research.

A. Arora is with The Ohio State University, Columbus, OH 43210. Phone: (614) 292-1836, email: anish@cis.ohio-state.edu

R. Jagannathan is also with The Ohio State University.

Y.-M. Wang is with Microsoft Research, Redmond, WA 98052.

is enriched, the dependability design may be modified accordingly.

### B. Rationale underlying model-based approach

Early techniques for system dependability assumed – with good reason– simple fault models, such as limited numbers of stuck-at faults, timing delays, or fail-stop nodes. During the last two decades many of these techniques (e.g., rollback-recovery, consensus, and voting) have been systematically extended to deal with a rich class of faults, such as any number of concurrent node failures and repairs, or Byzantine processes or arbitrary transient faults. Graceful degradation techniques have also addressed complex faults [1]. In one such technique, faults are decomposed in multiple fault-classes, and the system is designed to tolerate each fault-class in an appropriate manner. In recent work on component-based design of multitolerance [2], tolerances are added one at a time, with each step ensuring that the newly added tolerance component does not interfere with the previously added ones.

As may be expected, while techniques that deal with complex faults often yield elegant designs, there are cases where the added complexity has yielded substantial tradeoffs. ISIS is an example of a group communication services platform that dealt with complex faults but was itself so complex that maintainability suffered [3]. In other work, scenarios are reported where the virtual synchrony approach experienced scalability problems [4]. In the case of the Microsoft Cluster Service [5], it is reported that the high overhead of that dependability design prevented it from scaling past 2 to 4 nodes.

A different sort of scalability problem is noted for designs which assume full knowledge of implementation details: stabilization [6] with respect to the rich class of transient faults is a case in point. While there are cases where stabilization yields simpler designs than exception handling on a per-fault basis, it sometimes yields intricate designs because it makes extensive use of the details of system implementation. Also, the designs are rarely reusable for other systems because of their intimate dependence on the particulars of system implementation.

Experience suggests that to keep the design task tractable, the number and composition of the fault-classes should be kept well under control. Handling faults individually and efficiently is reasonable for common-case faults but not necessarily for rare-case faults (this is why we endorse fault-centric design for common-case faults). The added complexity in the latter case is a potential source of undependability. One alternative therefore is to group multiple rare faults together and deal with them collectively. Another alternative is to show that the rare faults are all members of a rich fault class, such as Byzantine faults or transient faults, and now tolerance may be designed more expeditiously with respect to that rich fault class. But in the light of the maintainability, scalability, and reusability problems discussed above, and with a view to deal with unanticipated faults as well, the alternative that we endorse is to deal with the complex faults in an implicit manner, by enforcing a model of the system.

### C. Outline of the paper

In Section II we describe model-based design in more detail. We illustrate our approach with two case-studies: a dependable lookup service in Section III and X10 powerline networking protocol monitoring in Section IV. Finally, in Section V, we make concluding remarks and discuss some directions for future work.

## II. Model-based Dependability

### A. Model

A *model* specifies the behavior that is desired of the system. This behavior may be expressed using any suitable modeling language, e.g., finite state automata, petri nets, temporal logic specifications, action systems, etc. Per se, a model of the system is independent of the faults that can affect the system. It is therefore assumed that the system satisfies the model when it executed in the absence of faults. By the same token, violations of the system model implicitly imply occurrence of some fault(s). The goal of the model-based approach, then, is to design dependability components that can be added to the system to “enforce” the model at all times, even in the presence of faults.

**Example:** Consider a highly available `HTTP` service. The service is identified by a unique domain name and is hosted by a cluster of workstations. It is assumed that in response to a request for name resolution with respect to this domain name, a `DNS` service supplies one of several internet addresses assigned to the cluster.

A model of the highly available service may be given with respect to the `IP` addresses: it is always the case that every `IP` address supplied by the `DNS` is assigned to some active node in the cluster. As long as the system satisfies this model the service remains available. Notice that the model is not specified in terms of the possible faults that may occur, e.g., NIC failure, workstation failure, unassigned `IP` address, duplicate `IP` address assignment, etc. □

### B. Model Predicates

The task of enforcement of a model is relatively straightforward for monolithic sequential systems. For instance, if the model is specified via an automaton, the model may be enforced by detecting/ensuring at each step that the system only performs valid transitions. But in distributed, concurrent, or composite systems, enforcement at the level of a global model of the system may be impractical and/or infeasible. For ease of implementation, therefore, we may decompose the model into several components (or processes or localities). The essential requirement of the decomposition is that every violation of the system model must imply the violation of at least one of the components models, and vice versa. It follows then that enforcement of the system model is achieved by separate enforcement of each component model.

Enforcing a (component) model is, without loss of generality, achieved by enforcing a corresponding "model predicate". While model predicates may, in general, involve temporal modalities, our presentation will focus only on state predicates of the model. This subclass of model predicates suffices in practice; moreover, by allowing the model state to be augmented with "history" or "prophecy" variables, the generality of this subclass is justifiable theoretically. Formally, then, we define a *model predicate* to be boolean expression on the state of the model. The state of the model comprises all variables existing in the model. Note that these variables exist only in the design space and are not necessarily implemented by the system. Borrowing notation from temporal logic, if $P$ denotes a boolean expression and $\Box$ denotes the *always* operator, then the model predicate $\Box P$ encodes the fact that the system always satisfies $P$.

**Example (contd.):** Continuing with our example of the `HTTP` server, we decompose the model into two parts and establish the corresponding model predicates: (i) each `IP` address is assigned to at least one active node in the cluster; and (ii) no `IP` address is assigned to two nodes simultaneously (which is a requirement of the underlying `IP` protocol). Letting $i, j$ range over the set of nodes and the model variables $up.i$ and $assigned.ip.i$ respectively denote that node $i$ is active and that the `IP` address $ip$ is assigned to node $i$, we express the above predicates as

$$\begin{array}{ll} Assigned: & \Box(\exists i : up.i \wedge assigned.ip.i) \\ Unique: & \Box\neg(\exists i,j : i \neq j : assigned.ip.i \wedge assigned.ip.j) \end{array}$$

Thus, there are two model predicates for each $ip$ address. Notice that the second predicate ($Unique$) can be further decomposed, on the basis of locality, into smaller predicates each of which refer to only two nodes. Whenever the model is violated, one of these predicates is also violated and, vice versa, whenever one of these predicates is violated the model is also violated. $\Box$

### C. Enforcement of Model Predicates

As discussed above, enforcement of all model predicates (corresponding to the component models) implies enforcement of the system model. But what precisely is meant by "enforcement"? Depending upon the level of dependability that is desired and the particular model predicate that is on hand, enforcement may mean any one of the following: (i) the predicate is never violated in the presence of faults; (ii) even if the predicate is violated, the system is restored such that the predicate is (re)satisfied; or (iii) violation of the predicate is notified in case the system functionality is unrecoverable; the notifications can be automatically logged for later analysis or propagated to the user for immediate action.

Enforcement of the individual predicates is achieved by dependability components which we term *predicate enforcers*. Achieving the level of enforcement in (i) above usually implies placing strict restrictions on the system functionality. Such predicate enforcers allow the system to make progress only if the next state can be guaranteed to still satisfy the predicate. In the case of (ii) and (iii) above, a generic strategy for the predicate enforcers is to monitor the corresponding predicate. If faults cause the predicate to be violated, the predicate enforcers execute appropriate corrective actions to resatisfy the predicate.

The corrective actions executed in the predicate handlers are not tied to any particular fault source or fault location, but only to the effects of the faults on the system and, in turn, the predicates. This empowers predicate enforcers to recover from the occurrence of unanticipated faults as well, unlike as in some fault-centric approaches where the handlers are tightly coupled with the source and location of the fault. In cases where the predicate enforcers are unsuccessful in restoring the system via the corrective actions, a notification is raised.

**Example (contd.):** Let us consider the enforcement of the predicate *Unique* extracted from our server model. The following predicate is for the `IP` address $ip$ and the two nodes $u$ and $v$ ( $u \neq v$):

$$\Box\neg(assigned.ip.u \wedge assigned.ip.v)$$

The level of enforcement we wish to provide for this predicate is as in (ii) discussed above. Now, if due to some fault, say in the DHCP (dynamic host configuration protocol) server, *ip* is assigned to the two nodes $u$ and $v$ at the same time and hence the predicate is violated, the enforcer for the predicate should take the corrective action to deassign the *ip* address from one of the nodes. Thus the predicate will be eventually resatisfied. $\Box$.

### D. Compatibility with Fault-Centric Design

In its pure form, model-based design does not model faults explicitly. However, we do not preclude the use of fault models in conjunction with the approach. The knowledge of fault models can be used to come up with efficient implementations for the predicate enforcers. In particular, a subset of the anticipated faults may be identified that are expected to occur more commonly than the others; a *common-case* fault model can therefore be defined explicitly. Using this fault model, it may be possible to ensure that some model predicates are never violated or more efficiently restored, even though this may not be possible in general for faults not in this fault model. This may be achieved by enforcing an additional predicate, that is stronger than the model demands. An illustration via our running example follows.

**Example (contd.):** The violation of the predicate *Assigned* from the server example implies that a particular *ip* has not been assigned to any alive node in the cluster. This may occur for example if the node to which the IP address was originally assigned fails, an occurrence which we treat as the *common-case* fault, as opposed to say simultaneous failure of two or more nodes, which we treat as a *rare-case* fault. A proposed enforcement of this predicate involves the failover of this address to some active node in the cluster. (We assume all the nodes are multi-homed, i.e., they are able to host more than one IP on the same NIC). It is possible to ensure that predicate *Assigned* is never violated for the common-case fault. In the implementation of the predicate enforcer, we use the common-case fault model to enforce the stronger predicate that at least two nodes are alive at all times:

$$\Box(\exists i, j : i \neq j : up.i \wedge up.j)$$

As long as this stronger predicate holds, even when a common-case single node failure occurs, there is at least one node alive in the system. This along with the failover of the IP addresses ensures that predicate $\Box(\exists i : up.i \wedge assigned.ip.i)$ is enforced without violation. $\Box$

### E. Comments on Model-Based Design

In model-based design, the model enables separation of the system functionality from the system dependability. This separation has the several advantages: (i) not only is the approach useful for adding dependability to systems during their design process but also a posteriori; (ii) in the absence of faults, the overhead of dependability is potentially limited to just monitoring that the system continues to satisfy the model; and (iii) the dependability components can be reused for systems that need to exhibit the same model of desired behavior.

The decomposition of the model into model predicates makes the design modular and incremental. If the system evolves or the dependability requirements change over time the dependability does not have to be always be re-implemented from scratch. We can selectively add new predicates or modify existing ones, and tailor the set of predicates to the new model. An couple of issues that arise in the implementation are, dependability of the predicate enforcers and possible interference among them. Discussion on these is deferred till after the case studies have been presented.

**Structure of Case-Study Presentations.** In both of the case-studies which follow, we begin with an informal description of the system, then give a model of desired behavior, and proceed to extract from the model the predicates to be enforced. Next, we discuss the common-case faults and a representative set of the rare-case faults, and finally present the design of the various predicate enforcers.

## III. Case Study: Dependable Lookup Service

Aladdin [7] is a system for dependable, extensible control of heterogeneous devices via an in-home PC cluster and heterogeneous network. Aladdin control scenarios include: (i) automatic device discovery and location mapping (e.g., plug a lamp into an outlet in the kitchen, turn it on, and the system will know that a new lamp is now available in the kitchen). (ii) natural language-based home automation (e.g., enter "turn on the lights on the garage side of the kitchen"). (iii) email-based remote home automation (e.g., send a secure email to close your garage door). And (iv) cell phone-based remote notification [8] (e.g., get a cell phone call when your basement is flooded).

Given the above scenarios, one of the keys for the extensibility of Aladdin is the *Lookup service*. This service responds to two types of queries: (i) an attribute based query which returns a list of unique names that match the attributes, and (ii) a name based query which returns the address of an object given the unique name. The lookup service maintains information regarding addressing, location, and status of the various types of objects in the home network, including sensors, devices, and controllers.

Objects typically join and leave the network spontaneously. To automate the discovery process, objects periodically "refresh" their status and location in the lookup service, at a frequency of their choice. This frequency is chosen based on a number of factors, e.g., how often their status changes and how much network bandwidth is available. Depending upon the frequency chosen, refreshes are classified as being either *low-frequency* or *high-frequency*.

### *A. Model and Predicates*

The requirements of the lookup service can be stated informally as: "each query to the lookup service returns a unique up-to-date response". For dependability, we replicate the service on an in-home PC cluster. Refresh messages and queries are assumed to be broadcast to all the replicas. The requirements of the lookup service may now be ensured by keeping the lookup server replicas in, say, virtual synchrony.

Alternatively, we could achieve lower-cost dependability by exploiting a model. We postulate the following model of the lookup service: Always there exists a unique server that responds to queries, and this server—which we refer to as the leader—has up-to-date status of the objects. If $i$ ranges over the replicas, the boolean model variable $alive.i$ denotes that node $i$ is running a functioning replica; $leader.i$ denotes that replica $i$ believes it itself is the leader; and $up-to-date.i$ denotes that replica $i$ has the most recent information. The table below lists the model predicates, extracted from the model.

**Lookup service predicates**

| | |
|---|---|
| Aliveness | : $\Box((\exists i : alive.i) \wedge (\forall i : leader.i \Rightarrow alive.i))$ |
| Uniqueness | $\Box((\exists i : leader.i) \wedge \neg(\exists i, j : leader.i \wedge leader.j))$ |
| Recency | : $\Box(\forall i : leader.i \Rightarrow up-to-date.i)$ |

### *B. Enforcement*

The dependability implementation, then, consists of enforcement of each of the three model predicates. Our implementation essentially (re)satisfies each predicate upon its violation. The enforcer for *Aliveness* runs a protocol that diagnoses and repairs failed replicas. The enforcer for *Uniqueness* implements a "weak leader election" protocol; in this protocol, only a unique replica knows it is the leader (the term 'weak' is used in the sense that the other nodes need not know the identity of the leader). Only the leader responds to queries.

*Recency* is automatically enforced in part by the periodic refreshes that are received from the various objects. The refreshes serve to repopulate the lookup database at the replicas once they have been repaired. While this is adequate for high-frequency refreshes, for the case of low-frequency refreshes the replica might respond to queries with out-of-date information for a long interval. Therefore, as part of the diagnosis and repair protocol, the information corresponding to the low frequency refreshes is streamed to the newly repaired servers to ensure *Recency* for low-frequency objects.

We identified the following faults as being common-case: the crash of a single replica and the loss of a single refresh message. To handle the crash of a single replica efficiently, the enforcer for *Aliveness* ensures that there are always two or more alive replicas with up-to-date information. In case the leader fails, the leader election protocol allows one of the remaining replicas with up-to-date information to automatically assume the role of leader. To handle the single refresh loss efficiently, we add an acknowledgment mechanism in which the leader is required to acknowledge the low-frequency refreshes. Low-frequency objects are expected to retransmit their refreshes until an acknowledgment is received. In case of high-frequency refreshes, high data-quality is achieved per se and so does not require the use of acknowledgments.

We conclude our lookup server case study with an example of an unanticipated fault that was handled by the predicate enforcers. During a deployment period of 14 days without any restart, on a cluster of four machines —Jasmine, MagicCarpet, Abu, and Genie— an unidentified 'glitch' caused unanticipated network partitioning on a number of occasions. We briefly describe one such occurrence (which in fact recurred several times): Magiccarpet become isolated from the rest of the nodes; Jasmine lost contact with Genie; and, Abu lost contact with both Jasmine and Genie. As a result of the partitioning, more than one node assumed leader status. In all cases, predicate enforcement restored the system to having a unique leader and subsequently the model was resatisfied.

## IV. Case Study: X10 Powerline Dependability

The X10 powerline network is one of several networks supported in Aladdin [7]. A typical X10 network consists of multiple controllers (CM11A interfaces) and multiple modules (LM465, PAM22 etc.), which communicate using the X10 protocol, over the common powerline communication medium. An X10 controller issues an address command, consisting of a house code and unit code (e.g., *A 1*), which places the respective module in an addressed state, and then issues a function command, with the same house code and a function code (e.g., *A On*). The addressed then responds to the function code specified as part of the function command.

The X10 protocol is a very simple communication protocol where the messages transmitted do not contain the identity of the transmitter (controller) or the receiver (module). The communication medium being a broadcast medium, it is not possible to distinguish the sequence of commands transmitted by any single controller. Moreover, the X10 modules attached to the powerline usually tend to be 'dumb' devices and hence it is not feasible to monitor each every device to detect if any fault has occurred. Therefore we elected to model the entire powerline network.

### A. Model

By considering the X10 messages as an input alphabet, we can model the network as an automaton that generates the sequence of transmissions of the powerline. A feature of the X10 protocol is that commands with one house code say $h$ do not affect the addressing or functioning of modules with a different house code. Therefore, the model automaton can be decomposed into independent automata one for each house code. The simplified automaton in Figure 1 models the legal sequences of X10 commands corresponding to house code $h$. For details regarding how the automaton was developed we refer the interested reader to [9]. The transitions in Figure 1 represent the valid system transitions and are labeled with the X10 commands observed/generated on the powerline. (All other transitions are invalid). The labels on the transitions are as follows: *AUOff(h)* — the `AllUnitsOff` function command; *ACmd(h,i)* — valid address command corresponding to an existing module ( house code $h$ and unit code $i$); and, *Brd(h)*, *Mul(h)*, *Uni(h)* refer to broadcast, multicast and unicast function commands respectively.

The legal sequences of X10 commands are governed both by the addressing logic and the function classification. The automata state $B$ represents the system configuration where no modules have been addressed and so only the broadcast commands can be issued. In states $Ui$, $UF$ we can execute unicast function `StatusReq` that polls the state of the appliance attached to the module. Since the response does not explicitly mention the address of the responding module we require only one module to be addressed so that response can be matched to the request. In the states $M$ and $MF$ where more than one module has been addresses only multicast commands are executed. The multicast commands, such as `On` and `Dim`, do not require a response so can be issued in these states also. The addressing logic governs the way the modules transition on the X10 commands.

**X10 powerline model**

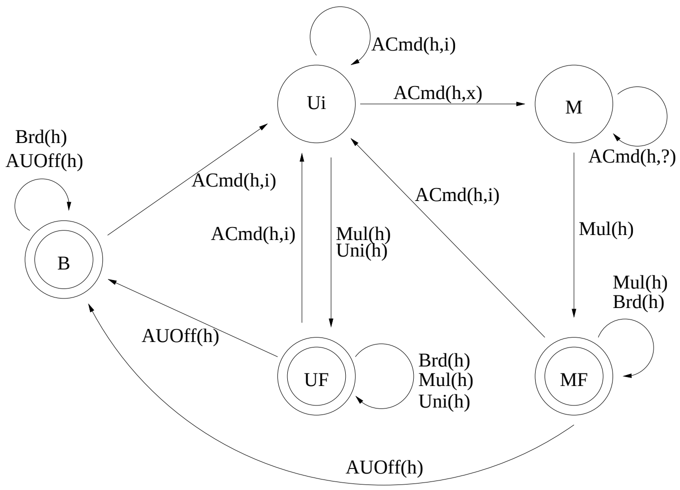


Fig. 1. Model automaton for house code $h$ ($x \neq i$, ? any unit code).

### B. Predicates

From the automaton model of the valid sequences, we extract the predicates that encode the valid transitions. If the model variable *state.h* abstracts the current state of the system and the model variable *cmd.h* abstracts the next command on the powerline, then for example the valid transitions from the state $M$ can be encoded as the predicate $(state.h = \text{ M }) \wedge (cmd.h = \text{ACmd(h)} \vee cmd.h = \text{Mul(h)})$. Alternatively, we can use the invalid transitions which are the complement of the valid transitions, to encode the same information. The advantage of this alternative is that we use the knowledge of the common-case faults to group the invalid transitions. A subset of the model predicates that are to be enforced are listed below. The first three predicates are self-explanatory.

**X10 powerline predicates**

Broadcast(h) :
$\Box\neg((state.h = \text{B}) \wedge (cmd.h = \text{Mul(h)} \vee cmd.h = \text{Uni(h)}))$
Unicast(h) :
$\Box\neg((state.h = \text{M} \vee state.h = \text{MF}) \wedge (cmd.h = \text{Uni(h)}))$
Allunitsoff(h) :
$\Box\neg((state.h = \text{Ui} \vee state.h = \text{M}) \wedge (cmd.h = \text{AUOff(h)}))$
Timeout(h) :
$\Box\neg((state.h = \text{Ui} \vee state.h = \text{M}) \wedge (cmd.h = \text{TO(h)}))$
ValidAddr(h) :
$\Box\neg(cmd.h = InvACmd(h))$

*Timeout(h):* The states $B, UF, MF$ are final states reached after a logical complete command sequence. The states $Ui$ and $M$, on the other hand, are states reached after a command sequence has been only partially issued. Therefore, in these states, we expect progress to be made in terms of the rest of the command sequence transmitted on the powerline. Any lack of progress is detected by using timeouts; the timer is started in the predicate enforcer, when states $Ui$ and $M$ are entered and the command $TO(h)$ captures the event that the timer has expired without any command having been issued. ( This is an event seen only in the predicate enforcer and not on the powerline ).

*ValidAddr(h):* The legitimate controllers on the powerline have knowledge about the valid X10 addresses that are actually assigned to modules. This information is updated in the lookup service on the addition of new modules. Given that $InvACmd(h)$ denotes an invalid address not currently assigned to any X10 module, the predicate *ValidAddr(h)* specifies that only valid addresses are to be transmitted on the powerline.

### C. Enforcement

The enforcement of X10 model predicates is complicated by the presence of "hidden" state. Unlike the lookup server example, where replicas can be polled to determine their status, the status of X10 modules (addressed or otherwise) cannot be determined directly. (The *StatusReq* command can be used to poll the status of the appliance attached to the module, but not that of the module itself). The system state is therefore hidden and must be deduced indirectly, from the sequence of commands observed on the powerline. We are therefore led to formulate the state deduction task in terms of the observability of the model, a concept which is well studied in discrete-event dynamic systems. We refer the reader to [9] for details.

The powerline medium is inherently unreliable and suffers from disruption due to power spikes and noise from household appliances. Its common-case faults are: the loss of a single message and the crash of a single CM11A interface. To detect and handle these faults, we propose that every PC be equipped with two CM11A interfaces: one serves as the controller through which the commands are issued, and the other to monitor the commands being transmitted. The software controller in the PC handles the loss of a single message by retransmitting it, and restores a crashed interface by resetting it.

All other faults are treated as rare-case and handled implicitly as predicate violations. Our current implementation of the enforcement generates a notification whenever a model predicate is violated. We present a few representative rare-case faults: (i) the crash of a the software controller that occurs in the middle of a command sequence leaves the system in states $Ui$ or $M$. This crash is detected when the timeout occurs in the predicate enforcer (TO(h)), causing the violation of the predicate $Timeout(h)$; (ii) if two controllers issue command sequences with the same house code, the resulting interleaved sequence could result in mismatched function commands. The predicates violated are `Broadcast(h)`, `Allunitsoff(h)` and `Unicast(h)`; and (iii) a security intruder without adequate information about the modules attached to the powerline could end up issuing an invalid address command, leading to violation of the predicate `ValidAddr(h)`.

We conclude our case study of X10 with a complex, unanticipated fault that we observed and which illustrates the value of the model-based approach. A transceiver module, which is supposed to convert radio frequency (RF) signals that it receives from wireless remotes into X10 commonds, once erroneously started to convert random RF noise (resulting from RF interference) into an invalid sequence of X10 commands.

## V. Concluding Remarks

In this paper we presented two case studies—a lookup service and an X10 powerline monitor—to illustrate the model-based approach. The lookup service study elicited the unanticipated fault of complex network partitioning. The X10 study dealt with the unanticipated arbitrary behaviors that resulted from RF interference. By dealing with all model-violations, the approach handled these and other unanticipated faults implicitly. We expect the approach to be useful in new application domains such as sensor networks and internet services where the environment is rather unknown and hence unanticipated faults are common.

As noted before, the approach does not preclude the use of explicit fault models. We chose to restrict ourselves to just two fault-classes –common-case and rare-case– but in general the designer is free to choose the number of classes as need be. In both case studies, for the common-case faults, the effect of the faults on the model behaviors was explicitly used in calculating the model predicates to be enforced. Rare-case faults were handled only implicitly and uniformly.

An analogy to intrusion detection is worth noting. Intrusion detection approaches are broadly classified as pattern-based and anomaly-based. Pattern-based approaches detect specific intrusions, which are analogous to our explicit handling of common-case faults. On the other hand, anomaly-based approaches start with a model of the ideal system and classify all deviations from the model as intrusions, which is analogous to our implicit handling of rare-case faults in terms of model violations.

By separating system dependability from system functionality and by decomposing a system model into predicates, the approach achieves modular (and incremental) design of dependability. Modular designs have been endorsed in other work: component-based design of multitolerance [2] achieved modular design using detector and corrector components. More recently, [10] illustrates the merits of separating dependability modules such as node failure detectors and arbitrary behavior violation detectors from functionality modules such as consensus builders in achieving distributed consensus despite some number of byzantine faults. Other examples of model-based modular designs may be found in [11], [12].

Modular designs of dependability lend themselves to object-oriented implementations. The predicate enforcer modules used in this paper involved monitoring of model predicates in order to detect their violations, which was sometimes event-driven —as in the case of X10 command transmissions— and sometimes time-driven— as in the case of periodic updates in the lookup service. The object model used in an object-oriented implementation therefore must support both types of monitoring. This may be achieved in terms of extant object models such as time-driven message-driven objects (TMO) [13].

Implementation of predicate enforcers also raises the issues of (i) interference with other modules and (ii) "who watches the watchers". The first issue involves interference not only with the underlying system but also the other predicate enforcers. This issue may be dealt with using existing existing concurrency control mechanisms. An alternative approach would be to use the semantic information encoded in the predicates to co-ordinate among the predicate enforcers at the semantic level; for a detailed discussion on how this alternative we refer the reader to [2].

The issue of the dependability of predicate enforcers arises since enforcers may be subject to the same faults that affect the system. In practice, this issue may not be as severe as the dependability of the functional components of the system, since enforcers are relatively simpler than functionality components. Also, enforcers are more under the control of the designer than are functionality components, and may be instantiated from carefully verified dependability frameworks. In principle, this issue may be dealt with by ensuring predicate enforcer dependability by reusing the model-based design approach. This was the approach taken in the case studies. The enforcers were designed to be themselves self-stabilizing and hence were in a position to deal with a number of rare-case faults, and they were also designed to tolerate the common-case faults efficiently.

**Future Work.** We envision the need for various services to support predicate enforcer modules. One of these is a core monitoring service. This service monitors allpredicate enforcers and restores them if they somehow fail. By requiring only this minimal level of functionality, the core can be designed to be highly dependable and also reused in various implementations. Yet another service would provide scalable and adaptive communication. This service would affect the overhead of monitoring of the predicate enforcers by the core, as well as that of the additional monitoring of the model predicates by the enforcers. Finally, another direction for future research is the automatic synthesis of model predicates and the corresponding predicate enforcers, which would simplify the design and implementation of dependability.

## References


[1] D. Siewiorek and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, Digital Press, 1992.

[2] A. Arora and S. S. Kulkarni, "Component based design of multitolerance," *IEEE Transactions on Software Engineering*, vol. 24, no. 1, pp. 63–78, 1998.

[3] K. P. Birman, "A review of experiences with reliable multicast," Tech. Rep. TR99-1726, Department of Computer Science, Cornell University, 1998.

[4] K. P. Birman *et al*, "The Horus and Ensemble projects: accomplishments and limitations," Tech. Rep. TR99-1774, Department of Computer Science, Cornell University, 1999.

[5] W. Vogels, D. Dumitriu, A. Agrawal, T. Chia, and K. Guo, "Scalability of the microsoft cluster service," in *Proceedings of the Second Usenix Windows NT Symposium, Seattle, WA*, August 1998.

[6] E. W. Dijkstra, "Self-stabilizing systems in spite of distributed control," *Communications of the ACM*, vol. 17, no. 11, 1974.

[7] 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.

[8] Y.-M. Wang, P. (Victor) Bahl, and W. Russell, "The SIMBA user alert service architecture for dependable alert delivery," in *International Conference on Dependable Systems and Networks (DSN 2001)*, Jul 2001.

[9] A. Arora, Y.-M. Wang, and R. Jagannathan, "Model based fault detection in X10 powerline monitoring," Submitted to *Eighth ACM Conference on Computer and Communications Security (CCS'2001)*. Tech. Rep., http://www.cis.ohio-state.edu/ anish/group/papers.html, 2000.

[10] R. Baldoni, J.-M. Helary, M. Raynal, and L. Tanguy, "Consensus in byzantine asynchronous systems," Tech. Rep. RR-3655, National Institute for Research in Computer Science and Automatic Control (INRIA), 1999.

[11] A. Arora, M. Demirbas, and S. S. Kulkarni, "Graybox stabilization," in *International Conference on Dependable Systems and Networks (DSN 2001)*, 2001.

[12] A. Arora, S. S. Kulkarni, and M. Demirbas, "Resettable vector clocks," *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 269–278, August 2000.

[13] K.H. Kim and C. Subburaman, "Dynamic configuration management in reliable distributed real-time information systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 239–254, 1999.