Integrated Verification for Robust Computing

Sanjit A. Seshia

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-103
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-103.html

July 31, 2006

Integrated Verification for Robust Computing

Sanjit A. Seshia
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720-1770
Email: sseshia@eecs.berkeley.edu

Abstract

This paper argues for an *integrated approach to verification*, combining offline and online verification techniques used at different stages of a system's lifetime, in order to detect and correct failures arising from design errors and program bugs. We propose a formal framework for online verification and recovery, including a fault model, within which such an integrated approach can be investigated.

As a particular instance, we consider the problem of recovering a class of finite-state systems, at run-time, from failures of safety properties, while leveraging results of offline verification. This class comprises systems whose behavior can be divided into rounds such that each round is largely independent of the others. We give a randomized recovery strategy based on online learning for which the expected number of recovery actions performed in any state is at most logarithmic in the total number of actions, assuming perfect error localization. Results from design-time verification can be used to reduce the search space of the recovery algorithm and deal with imperfections in error localization. We illustrate our approach with case studies.

Our results are a step towards building reactive systems that are robust to failures and self-evolve towards correct systems.

1 Introduction

Computer systems are ubiqituous today, and we rely heavily on them. However, much remains to be done to ensure dependable computing. In fact, improving the trustworthiness and robustness of computer systems has been identified as a grand challenge for Computer Science and Engineering [14].

System dependability can be improved by detecting and correcting errors. This can be done at various stages in the system's lifetime, ranging from design-time, through compile-time and install-time, to run-time. Accordingly, various techniques have been proposed that operate at each stage, such as design verification, static analysis, and run-time (dynamic) error detection and recovery. However, there is little, if any, communication and co-operation between the tools and techniques that operate at each stage.

We argue for an *integrated approach to verification*, wherein techniques and tools for detecting and correcting errors at various stages in a system's lifetime are combined to achieve robustness to failures at run-time. The focus is on errors that are introduced into the system by the designer/programmer and/or during synthesis/compilation.[1] We will refer to these errors as *design errors* or *bugs*. Pre-run-time techniques for detecting design errors, including both formal and informal methods, will be referred to as *design verification* or *offline verification*.

Our focus on design errors and the need for an integrated approach is motivated by the following factors:

[1]Some of the methods we propose in this paper could potentially be applied to other classes of errors.

1. **Limitations of offline verification:** The traditional approach to ensuring correctness of computer systems is *offline*, based on verification and testing systems before deployment. However, increasing complexity, short release cycles, and uncertain, dynamic environments are making it extremely hard to deploy systems that are devoid of design errors and program bugs. Many systems operate in unsupervised settings or under high-availability requirements, where "best-effort patching" of bugs does not suffice.

   We elaborate below on these challenges:

   (a) *Increasing complexity:* Consider the field of hardware design verification. In spite of significant advances, increasing design complexity is exacerbating the already high cost of design verification. As Bayazit and Malik [5, 34] point out, there is an exponential gap between the rate of increase of design complexity and the processing power available for verification. Furthermore, according to the 2004 update to the International Technology Roadmap for Semiconductors (ITRS) [2], verification is becoming the bottleneck in the hardware design process.

   (b) *Imprecise environment models:* The environment a system will operate within is often unknown before deployment; examples include space systems as well as Internet services. The traditional approach to deal with this in design verification is to adopt a conservative environment model. However, this can lead to a unacceptably high false alarm rate for the verifier, leading to decreased designer/programmer productivity.

   (c) *Evolving systems:* An increasing challenge for design verification is that computer systems of today are constantly evolving (and sometimes so are their requirements), with short deployment life-cycles. This is especially true of software, but with increased reconfigurability, it may soon be true of hardware systems as well. It may be unwise to spend huge resources on exhaustive verification of a design that will change soon unless the results of that verification effort can be re-used easily for ensuring that the evolving system stays robust to failures at run time.

   In light of these limitations, we cannot rely on offline verification alone.

   However, we stress that offline verification is an important step in ensuring dependability. Getting a system correct before deploying it, using exhaustive bug-finding or a correct-by-construction approach, is a goal worth striving for. The vast body of work on offline verification must be leveraged in any approach to dependable computing.

2. **Limitations of online verification and existing techniques for recovery:** *Run-time* or *online* verification offers a solution to some of the problems with design verification. In this approach, only those executions that occur at run-time are verified by a system component that monitors execution. Run-time monitoring has traditionally been hobbled by a high overhead cost, but recent work on hardware support for monitoring (e.g. [50]) have made this a promising alternative approach to design verification. Still, a question remains: when an error has been detected, how does the system recover from it?

   The most common traditional approach to recovery is based on periodic checkpointing, rollback on error detection, followed by a replay. This will not suffice for dealing with deterministic errors, as the error will simply re-appear on replay. The existing solution to this problem is to use for replay a different, simpler version of the system that must be proved correct at design time. This approach has the following shortcomings:

- A simpler design is (typically) slower than the optimized version, so it can lead to substantial overhead, especially for high performance applications.
- Formally verifying the correctness of the simpler design itself is not necessarily easy.
- For software, programmable systems, and arbitrary hardware circuits, it may not be easy or cost-effective to manually come up with a substantially simpler design.

One can also view this traditional approach as a form of "n-version programming", which previous studies have indicated to have a high cost of development and maintenance with unclear benefits for system dependability [31].

Thus, online verification on its own does not suffice to ensure system dependability.

The thesis of this paper is that the challenges to both offline and online verification can be overcome by using *an integrated approach to verification*, wherein offline and online verification are combined to achieve not only efficient run-time monitoring of systems, but also *online recovery*. In particular, we propose a new formal approach to online verification and recovery based on leveraging (even partial) results of offline verification. We are concerned in this paper with *reactive systems*, i.e., systems that run forever and interact with the environment at the speed of the environment.

We propose and investigate the following new research directions:

1. *A formal framework for run-time recovery from design errors.* Current recovery methods are limited to known classes of bugs (e.g., buffer overruns) and there is no formal framework within which their pros and cons can be analyzed. In this work, we seek to create such a framework, leveraging the extensive prior work on formal models of reactive systems.

2. *Learning to optimize recovery.* The run-time overhead of recovery can be mitigated by self-evolving the system over time, by use of online and offline learning. The proposed work seeks to utilize, specialize, and extend prior work in learning theory for this purpose, for instance, in the model of *learning with mistake bounds*.

3. *Leveraging offline verification.* We investigate ways to leverage *partial results* of offline verification in order to reduce the overhead of run-time verification and improve the effectiveness of the recovery.

4. *Quantifying dependability with a fault model.* It is essential to gain a quantitative understanding of the trade-off between spending resources on offline verification and incurring an overhead at run-time. Toward this goal, we are developing formal models of faults leading to design errors.

The primary offline verification technique considered in our work is *model checking*. Model checking is perhaps the most widely-used automated formal verification technique available today, for both hardware and software. However, perhaps the most frustrating aspect of model checking is that, given short design cycles, the process often does not terminate with a yes/no answer within the time allocated for verification. In such cases, one is left with little to show for the many hours (or days!) for which the model checker was run.

Techniques for run-time verification and monitoring are not the focus of this paper; there is a large and active body of work in that area which we seek to build upon.

The rest of the paper is organized as follows. Section 2 starts with some basic background terminology.

We introduce basic ideas in our formal approach in Section 3. Ideas are explored in the context of a class of finite-state systems whose behavior exhibits a form of *state-renewal*, wherein each execution can be divided into rounds such that each round is largely independent of the others. We define a fault model for finite-state systems that covers common classes of faults introduced by designers/programmers, and formally define what critical system functionality may be preserved during recovery.

We then describe in Section 4 how system recovery can be optimized by use of learning. In particular, we give a randomized strategy the system can follow so that the expected number of recovery actions performed for any fault is at most logarithmic in the total number of actions, assuming perfect error localization.

Section 5 shows how the presence of even partial offline verification results helps improve the proposed recovery procedure. Results from offline verification can be used to reduce the search space of the recovery algorithm and deal with imperfections in error localization.

A case study is described in detail in Section 6. Related work is surveyed in Section 7. We conclude in Section 8 with a discussion of the paper's contributions and directions for future work.

2 Background

We use the taxonomy for dependable and secure computing given by Avizienis et al [4].

A *system* is an entity that interacts with other entities (systems), including hardware, software, human beings, and the physical world. These other systems form its *environment*.

The focus of this paper is on making systems robust from errors that the designer/programmer might introduce into the system description, especially those errors that are deterministic in nature. We assume that the system is accompanied by a formal specification of its correct behavior, as elaborated in Section 3.

A *failure* occurs in a system when its behavior deviates from the specification. The part of the system state that exhibits this deviation is the *error*. The adjudged or hypothesized cause of an error is called a *fault*.

3 Formal Framework

We now give a formal treatment of the problem of recovering from run-time failures of a class of finite-state systems. We begin in Section 3.1 with the formal model of this class of systems along with relevant definitions. A fault model is introduced in Section 3.2 as a way of quantifying the capabilities of a recovery technique.

3.1 System, Properties, Error Detection, and Recovery

The system $\mathcal{M}$ is modeled as a finite-state transition system, represented as a tuple $(\mathcal{S}, \mathcal{A}, \delta, I)$, where

- $\mathcal{S}$ is the set of system states;
- $\mathcal{A}$ is the set of system actions;
- $\delta \subseteq \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function that describes the next state of the system that results from performing an action in the current state; and
- I is the set of initial states of the system.

The above system model is standard. Formally, an action is a predicate over two states, and can be represented as a Boolean expression containing current and next state variables. Examples of actions include system calls, message sends and receives, incrementing a hardware counter, setting a bit, etc.

In addition to being finite-state, the system obeys a *state-renewal* property. Specifically, after startup and initialization, the system *behavior* (also termed as *run* or *execution*) can be divided into *rounds*, each of which begins in a "valid" start state and is of finite, but arbitrary, length. A valid start state is defined by a state invariant I_{start}. This is typical of many reactive systems, whose runs are infinite, but are composed of terminating sub-computations performed within a non-terminating sense-response loop.

Formally, each minimally correct behavior (trace) of the system must be a sequence of states and actions taking the form

$$s_0\ a_1\ s_1\ a_2 \ldots a_{i_1}\ s_{i_1}\ a_{i_1+1}\ s_{i_1+1} \ldots a_{i_2}\ s_{i_2}\ a_{i_2+1}\ s_{i_2+1} \ldots a_{i_3}\ s_{i_3}\ a_{i_3+1}\ s_{i_3+1} \ldots\ldots$$

where

- $s_0 \in I$;
- $s_i = \delta(s_{i-1}, a_i)$;
- $\forall j \in 1, 2, 3, \ldots, s_{i_j} \in I_{start}$.

The finite sub-trace starting at s_{i_t} and ending before $s_{i_{t+1}}$ is termed as *round* t. We make a distinction here between the initial state of the system (s_0) and the state in which the system starts its state-renewal behavior (s_{i_1}) to model system "boot-up."

The reason we require the state-renewal behavior is so that errors do not accumulate to the extent that the system becomes unrecoverable. By requiring the system to return to a valid start state in each round, we place a minimal correctness requirement on each execution.

We note that each round is "largely" independent of the other in that the system begins each round in a valid start state. An example of a system exhibiting this behavior is a network packet processing system performing a task such as a packet forwarding: whether and where a packet is forwarded is usually independent of how its predecessors were processed. Similarly, in reactive programs that operate in a sense-response loop, the start of a new iteration corresponds to the program returning to the head of the loop, so I_{start} is simply a predicate on the program counter.

Note that even systems that do not exhibit the above behavior could potentially be viewed in this way. Each round could be a finite, but arbitrary-length prefix of a run of the system that prematurely ends in failure, with system reboot leading to the start of a new round. We can model aspects of recovery-oriented computing [10] approaches in this manner.

Note that we make no assumptions about the environment, in particular about whether it is finite-state or otherwise.

System Description. The transition function of the system can be succinctly represented as a *guarded command program*. This is done by viewing each action a_i as a *guarded command* of the form $g_i \rightarrow op_i$, where g_i is a Boolean expression over the state variables of the system and op_i is the assignment to state variables corresponding to a_i. We will term op_i as the *operation* corresponding to a_i. Thus, $g_i \rightarrow op_i$ defines operation op_i to be taken in all states that satisfy the guard g_i. We assume the guards to be mutually exclusive and together they exhaustively cover all system states.

Finite-state systems implemented in languages such as C and Verilog can be translated into the above form. For instance, for a C program, a guard will include the corresponding program counter location and any conditionals under which an action is taken (statement is executed).

3.1.1 Properties and Recovery

The goal of recovery is to preserve "critical system functionality" when an error has occurred or is imminent.

For this section, we will formalize critical functionality of system $\mathcal{M}$ in the tth round of operation as the combination of the following properties:

1. $\mathcal{M}$ must return to a valid start state for the next round $t+1$ ($s_{i_{t+1}} \in I_{start}$; and

2. $\mathcal{M}$ must satisfy a *safety property* (e.g., all states must obey an invariant property).

For this paper, we further restrict the first property to be a bounded liveness property by requiring the system to return to a valid start state within a specified number of steps of the previous start state. This is often the case for reactive systems where sub-computations in each round are guaranteed to be terminating (e.g., embedded systems composed of tasks satisfying real-time constraints).

The simplest form of the second property is an *invariant*: a Boolean formula over state variables expressing a condition that each safe system state must satisfy. We will express such a property in terms of its negation, the error predicate $\mathcal{E}$, that defines error states.

We deal with deadlock by modeling absence of it as a safety property.

In some cases, the second property can be relaxed further to allow for a "few violations" of the safety property. For instance, in a network processor, it might be acceptable for a few packets to be misforwarded, but the fraction of these must be vanishingly small. We will see an example of this in the case study described in Section 6.

$\mathcal{M}$ is *correct* if every execution preserves critical functionality as defined above, where the safety predicate is defined as $\neg\mathcal{E}$. Otherwise, we say that $\mathcal{M}$ is *buggy*.

A *move* is an operation that the system $\mathcal{M}$ has full control over. A failure occurs in a round of system $\mathcal{M}$ because $\mathcal{M}$ performs one or more incorrect moves during that round. We will term such an incorrect move as an *error move*.

A *recovery strategy* replaces each error move with an alternative *repair move*.

We will sometimes refer to the problem of devising and executing a recovery strategy as the system *repair problem*.

Finally, we mention that an online error detector can run in one of two modes:

1. *In lockstep with the implementation or with a slight look-ahead, detecting an impending failure one or more steps before it occurs.*

   For instance, suppose system $\mathcal{M}$ is in state s and about to perform action a. We can detect whether an error is imminent by checking if the next state $s' = \delta(s, a)$ satisfies $\mathcal{E}$.

   The goal of recovery here is to steer the system away from the error state by modifying its transition function.

2. *In a slightly delayed mode, where rounds of execution of the implementation that have already occurred are analyzed by the detector for failures.*

   This can be used in applications where a few failures can be tolerated (such as network processors dropping a few packets in error). The goal of recovery in this case is to automatically evolve the system towards correctness over several rounds of execution.

3.2 Fault Model

A fault model defines the capabilities of a recovery technique in terms of exactly what kinds of faults (errors) can be detected and corrected. It plays a similar role to error models in coding theory that specify, for example, the number of bit flips or erasures that an error-correcting code can detect and correct.

Suppose that we are given a finite-state transition system $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \delta, I)$. As mentioned earlier in this section, the transition behavior of $\mathcal{M}$ is represented as a guarded command program

$$\begin{array}{ccc} g_1 & \rightarrow & op_1 \\ g_2 & \rightarrow & op_2 \\ & \vdots & \\ g_k & \rightarrow & op_k \end{array}$$

where operations $op_1, op_2, \ldots, op_k$ are distinct, the guards are mutually exclusive ($g_i \wedge g_j = \mathbf{false}$ for $i \neq j$), and together they exhaustively cover all system states ($\bigvee_i g_i$ is a tautology). We will assume that, without loss of generality, g_k is a special *default* guard that covers all states that the remaining guards do not include (this is commonly the case in system descriptions). Note that the total number of guarded commands is k.

As mentioned in Section 3.1.1, a failure in a round of system execution is caused due to a sub-sequence of one or more error moves. The error model we introduce here specifies what kinds of sequences can be detected and corrected.

3.2.1 Kinds of Errors

We first define the kinds of errors that a designer/programmer might introduce into the system description. Assuming that the set of initial states I is specified correctly, all errors must be in the guarded command program. These can be of the following kinds:

1. *Substitution errors:* A guarded command program has a substitution error if one of its actions $g_i \rightarrow op_i$ must be substituted by a new action to make $\mathcal{M}$ correct.

2. *Insertion errors:* A guarded command program has an insertion error if a new action $g_i \rightarrow op_i$ must be inserted to make $\mathcal{M}$ correct.

   To avoid introducing non-determinism into the system, the guards in the program may need to be adjusted after a new action is inserted into it. (If the new guard is mutually exclusive with the other non-default actions, no adjustment is required.) Thus, an insertion performed to correct an error might also result in substitutions.

In general, a buggy system may have one or more substitution and insertion errors. We have found this class of errors to be sufficient for our preliminary work and to model errors reported in the literature. For instance, Groce et al. [24] discuss the "root causes" of errors found by the CBMC model checker [11] in source code for two programs. The first is the Resolution Advisory component of the Traffic Alert and Collision Avoidance System (TCAS) included in the Siemens suite [41]. The second is a version of the μC/OS-II real-time kernel [3] source code. The former has a substitution error, where a conditional (guard) is wrongly specified. The latter has an insertion error, where a return statement was missing at a certain line in the code.

In this paper, we discuss repair strategies for substitution errors where only the operation part of the action needs to be substituted. Dealing with other classes of errors is left to future work.

We denote the number of such substitution errors in a guarded command program by N_{err}.

3.2.2 Elements of a Fault Model

A fault model has three parameters:

1. N_{emove}: The maximum length of a sub-sequence of error moves that can be corrected.
2. N_{steps}: The number of actions (steps) from the last error move within which any failure can be detected.
3. N_{err}: The number of substitution errors in the guarded command program representing system $\mathcal{M}$.

The first parameter is a measure of how complicated a failure the system can self-correct; a failure that results from a long chain of error moves (a large value of N_{emove}) is likely to be hard to debug and fix, even manually. The second parameter is a measure of how precisely the run-time verifier can localize the error. Smaller the value of N_{steps}, better the error localization. The third parameter is a measure of how buggy the system $\mathcal{M}$ is.

4 Optimizing Recovery with Learning

The overhead of needing to recover from failures in a round can be mitigated by incorporating a form of self-repair into the system, where the system learns from failures in previous rounds of execution. In this section, we consider how the online recovery process can be optimized by use of *online learning*. In particular, one can view the online recovery problem as one of *learning from mistakes*. This allows us to leverage the work in computational learning theory under the *mistake bounds model* [32]. System rounds correspond to the "trials" in which learning proceeds.

For ease of presentation, we initially consider an fault model in which $N_{emove} = 1$ and $N_{err} = 1$; i.e., there is only one substitution error in the original guarded command program and the recovery strategy can recover from a failure caused by a sequence of error moves of length one. Even this highly restricted fault model brings out many interesting aspects of recovery. The restriction is lifted in Section 4.4, with a discussion of the resulting implications.

4.1 A Simple Scenario

We first consider the simplest situation where $N_{steps} = 1$ (in addition to $N_{emove} = 1$ and $N_{err} = 1$). In other words, we have perfect error localization.

Suppose the system arrives at state s and is about to perform the move corresponding to non-input action a_1. Suppose that the run-time monitor detects that the imminent next state $s' = \delta(s, a_1)$ satisfies $\mathcal{E}$, i.e., s' is an error state. We must now choose a new move to perform in state s. The question is: which one?

The situation is depicted in Figure 1. The currently defined transition out of state s is shown as a solid line. In order to avoid the failure, the system $\mathcal{M}$ must replace the operation associated with its current action a_1 by a new one. Suppose that the options for replacement result in alternate actions $a_2, a_3, \ldots, a_m$. By evaluating $\delta(s, a_i)$ for each i, we can select a subset $\mathcal{A}_r$ of these alternative actions that will avoid the error

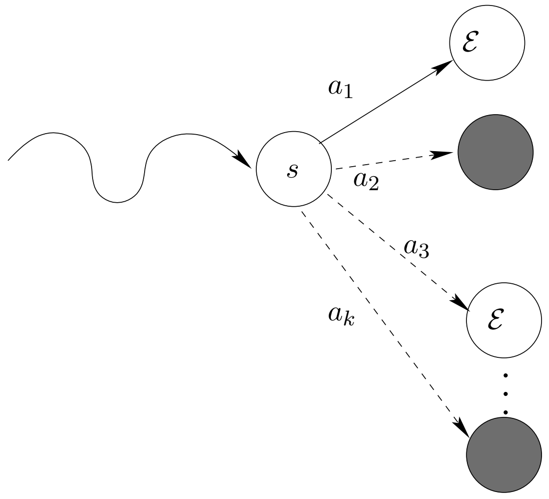

Figure 1: **Recovering from an error with perfect error localization.** a_1 is the currently defined action in state s and $a_2, a_3, \ldots, a_k$ are the alternatives.

in the next step. In Figure 1, the alternative actions that avoid the error are shown by indicating their target state in gray.

Note that set $\mathcal{A}_r$ forms the search space for the recovery strategy. This space can be very large. In fact, the cardinality of $\mathcal{A}_r$ can be $O(|\mathcal{S}|)$, the number of system states. This is because each alternative action can correspond to a transition to a unique system state.

Consider using the following *greedy* strategy:

> *Greedy:* Pick an arbitrary action a_r from $\mathcal{A}_r$, according to a deterministic heuristic, and replace a_1 by it.

By construction of $\mathcal{A}_r$, the resulting guarded command program will avoid entering an error state in the next step of the current round of execution. However, the following problems can still occur:

1. *Problem:* The system might encounter an error state in two or more steps, within the current round.

   Reason: The chosen action a_r is incorrect. By assumption, $N_{err} = 1$, so a different action cannot be the cause of the error. However, it is possible that using a_r led the system into a part of the state space that should not be reached under correct execution, and the error was then encountered on an action different from a_r. Even in this latter case, the cause of the imminent failure is a_r.

2. *Problem:* The system might encounter an error state in a subsequent round of execution.

   Reason: The chosen action $a_r = g_r \rightarrow op_r$ is incorrect. Even though a_r avoided the error for the environment inputs provided in the current round, the input received in the subsequent round led the system to a state satisfying g_r from which performing a_r will result in a failure.

In either case, we can conclude that if the system encounters an error state, it must be the case that a_r is an error move and must be replaced by an alternate action.

The lesson is that a greedy recovery strategy can have a high overhead, with recovery actions required in several rounds of execution. In fact, if the recovery strategy is greedy, the system might be required to perform an unbounded number of recovery steps in its lifetime, even when there is a valid fix to the substitution error. This is because the environment could alternate between two inputs inp_1 and inp_2 that causes the system to toggle between two repair actions that lead to errors on inp_1 and inp_2 respectively.

4.1.1 Use of Randomization

One of the problems with the greedy strategy is that the choice of repair action a_r is deterministic. What if we randomized this choice?

More precisely, suppose we follow the following repair strategy:

Uniform Random: Pick an action a_r from $\mathcal{A}_r$ uniformly at random, and replace a_1 by it.

Denote the cardinality of $\mathcal{A}_r$ by m and the number of *correct fixes* (actions that make $\mathcal{M}$ correct when substituted in for a_1) in $\mathcal{A}_r$ by m'. Then, the probability of picking the correct fix in a single trial is $p = \frac{m'}{m}$.

The uniform random (UR) strategy faces the same two problems as the greedy strategy. However, if there is at least one correct fix, the expected number of recovery actions that UR performs before finding the correct fix is

$$\sum_{i=1}^{\infty} i \cdot p \cdot (1-p)^{i-1} \;=\; \frac{1}{p} = \frac{m}{m'} \tag{1}$$

In the worst case, $m' = 1$, and thus the expected number of recovery actions that the UR strategy takes is m.

While better than the greedy strategy, this is still not good enough. In the worst case, the number of actions is equal to the number of states of $\mathcal{M}$, which can be exponential in the number of state variables, a fairly high overhead!

4.1.2 Use of Learning

The problem with both greedy and UR strategies is that they do not learn from past *mistakes*. Depending on the mode of online error detection (as introduced in Section 3.1) a mistake can either be (1) a recovery action that is not the correct fix (if one exists), or (2) a failure of the implementation. In either case, we want to obtain a recovery strategy that minimizes the number of mistakes it makes.

Mistake Bounds and the Experts Problem. A natural setup for this problem is to treat it as one of learning under a mistake bounds model [32]. Conceptually, there is one learning problem for each substitution error encountered during recovery.

An existing learning problem that fits our scenario well is the *experts problem* (also known as the problem of predicting from experts advice) [7, 33]. The problem is best stated as a multi-round game as follows.

- *Setup:* There are two players whom we will refer to as the *protagonist* and the *antagonist*. In addition, there are n experts who make a binary prediction in each round.
- *Play:* At the start of each round, the protagonist must choose an expert to "go with," while the antagonist picks the "right answer" for the predictions. However, they make their selections unaware of each other's choices. At the end of the round, the choices are revealed.

  If an expert predicted the antagonist's choice incorrectly, she incurs a cost of 1, else 0. The protagonist incurs the cost of the expert he chose.
- *Goal:* The goal for the protagonist is to use a selection strategy that incurs a cost as small as that incurred by the best expert, in hindsight, over several rounds.

Suppose recovery is initiated in round t at state s_i satisfying guard g_i. Let Γ_i denote the set of all moves that can be performed in s_i.

The repair problem considered in this section is mapped to the experts problem as follows:

- Experts correspond to elements of Γ_i, i.e., candidate repair moves as well as the current move in s_i. The system $\mathcal{M}$ is the protagonist and its environment is the antagonist.
- Each move γ_i in Γ_i is evaluated in s_i to ascertain whether it will result in a failure. If so, the move "predicts" that it will result in an error the next time recovery is initiated in some state s_i' satisfying g_i, otherwise, not. The next time for recovery could occur either later in round t or in a subsequent round.
- The system $\mathcal{M}$ must select one of the moves in Γ_i, denoted by γ_r, as a repair move to "go with". In other words, $\mathcal{M}$ will modify its transition behavior (for the rest of round t and all subsequent rounds) by using γ_r in all states satisfying g_i.
- The environment selects inputs to $\mathcal{M}$. If some candidate repair move γ can result in a failure, it means that the environment is able to supply the input that will trigger that failure. Thus, the environment's choices decide which candidate repair moves will result in a failure the next time recovery is initiated.
- The goal of $\mathcal{M}$ is to use a recovery strategy that incurs as low a cost as the best candidate repair move would have done, in hindsight. If there is a correct fix, we would ideally like $\mathcal{M}$ to select that fix rightaway and incur 0 cost after the first recovery.

Remark 1 There is one potential problem with the above mapping. If $\mathcal{M}$'s choice of γ_r is observable by the environment, it can always force $\mathcal{M}$ to fail. Therefore, we assume for this paper that the environment is oblivious to $\mathcal{M}$'s choice, at least until that choice results in a failure. Relaxing this assumption will be the subject of future work.

We have been able to formulate the problem of selecting a recovery move as an instance of the experts problem. The next question is: how do we solve it?

The Weighted-Majority Algorithm. Littlestone and Warmuth gave the *weighted majority algorithm* (WM) to solve a version of the experts problem [33]. Later, Freund and Schapire [23] showed how to use a modified, randomized version of that algorithm (RWM) as a way of solving the experts problem. RWM makes at most $O(\log n)$ mistakes (assuming there is a correct fix), where n is the number of experts, but its running time and space, per round, is $O(n)$. For a proof of this result, see the above mentioned papers [23,33] or Blum's survey paper [7].

The randomized weighted-majority (RWM) recovery strategy is given below:

Randomized Weighted-Majority:

- Initially, all experts (recovery moves) are assigned a weight of 1.
- When an expert makes a mistake, the associated weight is multiplied by a factor $\beta \in (0, 1)$.
- (randomized step) At any time, the algorithm picks the expert with probability equal to that expert's fraction of the total weight.

If set of candidate repair moves is of size m, this means that using this algorithm will incur an expected number of $O(\log m)$ mistakes. This is true even if there is only one correct fix amongst the m alternative actions, unlike the case of the UR strategy. The overhead imposed by RWM seems acceptable, given that $\log m$ is of the order of the number of state variables.

Thus, the use of online learning can be very beneficial in reducing the overhead of a recovery strategy, even in the simple case of $N_{steps} = N_{emove} = N_{err} = 1$.

However, the run time of the RWM recovery strategy is $O(m)$. Since m can be as large as $|\mathcal{S}|$, the run-time and space requirements of RWM can be too high for a practical implementation, unless restrictions are placed on the kinds of repair actions to be considered. Techniques to mitigate this overhead will be explored in Section 4.3.

Remark 2 If a correct fix does not exist, the RWM strategy guarantees that in the expected case $\mathcal{M}$ will make only $O(\log m)$ more mistakes than if it started and stuck with the recovery move that made the fewest mistakes in hindsight.

Remark 3 Note that the mapping to the expert problem given above is not the only one possible. Domain-specific heuristics can be used as "expert advice" and the RWM strategy can be used to do almost as well as the best domain-specific heuristic will.

4.2 Dealing with Arbitrary N_{steps}

We now relax one of our assumptions, allowing $N_{steps} \geq 1$; i.e., the failure might be detected more than one step after the error move was made.

Consider round t of operation of $\mathcal{M}$ that results in a failure, producing the following error trace:

$$s_{start} \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots s_{n-1} \xrightarrow{a_n} s_{err}$$

where s_{start} is the start state for round t, and $s_{err} = s_n$ is an error state, i.e., one satisfying the error predicate $\mathcal{E}$.

On failure, $\mathcal{M}$ initiates recovery by rolling back at most N_{steps} steps and steers itself into a non-error state, if one exists. (If one does not exist, it means that there is no correct fix to the guarded command program.) The steering is performed by replacing a_i for some i, where $\max(1, n - N_{steps} + 1) \leq i \leq n$, by an alternative action a_r. In this case, the recovery strategy must make two selections:

1. A state s_i at which to replace a_i; and

2. An alternative recovery action a_r to take in s_i: a_r is a candidate alternate action if replacing a_i with it in the guarded command program and replaying the execution from s_i avoids the error state s_{err}.

In fact, we need to replace an action in the guarded command program (GCP) representation of $\mathcal{M}$. For a GCP with k actions, we need to roll back at most $N_{gcp} = \min(k, N_{steps})$ steps. Therefore, the search space for a repair move to correct for a single failure is

$$m \cdot N_{gcp} = m \cdot \min(k, N_{steps}) \tag{2}$$

where, as before, m is the number of alternate recovery moves (actions) to be consider at each step.

Let us first consider the asymptotic running time of our recovery algorithm. The number of candidate repair moves to be considered, in the worst case, is $O(m \cdot N_{gcp})$. Assuming a cost model where each step incurs the same cost, the time taken to check whether a candidate repair move avoids state s_{err} by replaying with the repair move is $O(N_{steps})$. Finally, denote the time taken to select one of the candidate repair moves that avoids s_{err} by T_{rec}, the time taken by the recovery strategy.

If we use RWM to select a repair move, T_{rec} is $O(m \cdot N_{gcp})$.

Thus, the overall run time is $O(N_{steps} \cdot m \cdot N_{gcp})$.

This is clearly unacceptably high in the worst case, since m can be as large as $|\mathcal{S}|$.

4.3 Practical Concerns

The time and space requirements of the recovery strategies discussed above can make them impractical. In this section, we briefly discuss some of these problems and potential solutions.

4.3.1 Backtracking and Search Overhead

A source of overhead is the linear dependence of the run-time on the number of candidate repair moves, m.

A natural approach is to work with a restricted set of potential repair moves Γ_r. Ideally, one would like this set to have cardinality exponentially smaller than the number of states in the system. Dealing with defining such a set is left to future work. Note, however, that this restriction is easily done in a "design-for-recovery" setup, where the system designer specifies an exhaustive set of error moves. In that case, Γ_r is typically a $O(1)$ sized set that is statically chosen.

Exploiting parallelism is another approach that can be used to mitigate the impact of having many candidate repair moves. The different options for recovery can be explored in parallel as long as the corresponding state is maintained disjoint from each other and without any global effects.

4.3.2 Recovery Strategy Overhead

RWM requires that we maintain weights for each candidate repair move. In other words, it requires $O(m)$ space for each learning problem, which can be excessively high. This can be addressed by using a restricted set of candidate repair moves, as mentioned above.

However, it is likely that most of the candidate repair moves will avoid the error in most states. This is because operations typically modify a small subset of the overall state and are thus likely to preserve the safety property. Thus, only a small subset of weights would need to be adjusted on each round of the RWM algorithm, and it is worth investigating efficient implementations for this "sparse" case. One way of exploiting sparse updates is to maintain a cache of weights rather than a full table.

Maintaining weights involves floating-point computation, which can be expensive. The parameter β can be chosen as $\frac{1}{2}$ to reduce the overhead of floating-point division by forcing weights to a power of 2. If the weights are maintained as powers of 2, they can be maintained as integers and in fact can be stored compactly using Bloom filters [6] or their counting variants.

In summary, we believe with a suitable combination of implementation techniques, the overhead of RWM can be reduced.

4.4 General Case

At the start of this section, we assumed that $N_{emove} = 1$ and $N_{err} = 1$. We now relax these assumptions and consider the implications. As before, we consider an error trace of the form

$$s_{start} \xrightarrow{a_1} s_1 \xrightarrow{a_2} s_2 \ldots s_{n-1} \xrightarrow{a_n} s_{err}$$

where the failure was detected within N_{steps} of the last error move.

First, consider what happens if $N_{emove} \geq 1$, but $N_{err} = 1$. This means that the failure could have been caused by a chain of at most N_{emove} error moves, but each of those is due to the same substitution error in the guarded command program.

Since there is only one error that needs fixing, this case folds to the same situation as in Section 4.2. The recovery step must try, for each step in the trace that corresponds to a unique guarded command program (GCP) action, an alternate repair action. The number of such steps can be at most $N_{gcp} = \min(k, N_{steps})$. As before, the search space for recovery is of size $m \cdot N_{gcp}$.

However, the run-time can be larger, since one might need to backtrack more than N_{steps} steps to replay. (Note that N_{steps} is the number of steps since the *last* error move.) In the worst case, one might need to backtrack n steps to s_{start}.

Thus, if $N_{err} = 1$, the run-time for recovery is $O(n \cdot m \cdot N_{gcp})$.

However, the situation is considerably harder if $N_{err} > 1$. Let us consider the case where $N_{err} < N_{emove}$, and both can be greater than 1.

A recovery strategy must now consider all possible sub-sequences in the above trace of length N_{emove}. For each sub-sequence, at most N_{err} of the moves that correspond to distinct GCP actions must be replaced by alternative repair moves. Thus, it must pick N_{emove} positions out of the at most n positions in the above trace, and for each substitution in the GCP, it must select from the available candidate repair moves. The size of the search space is therefore

$$ {}^{n}P_{N_{emove}} \cdot m^{N_{err}} \tag{3}$$

where the first term corresponds to picking N_{emove} positions and the second term corresponds to searching for a combination of error moves to use in those positions.

Clearly, the main implication of relaxing the $N_{err} = 1$ assumption is a combinatorial explosion in the search space that the recovery strategy must consider, directly impacting the time and space overheads of the recovery strategy (e.g., RWM). The overall run-time for recovery, factoring in the time for replay, is $O(n \cdot {}^{n}P_{N_{emove}} \cdot m^{N_{err}})$.

In Section 5, we explore the use of offline verification, even with partial verification results, to reduce this search space.

5 Leveraging Offline Verification

Practically all computer systems undergo some form of verification before deployment, by way of testing, simulation, static analysis, and/or formal verification. Verification is an integral part of the product design cycle. However, in spite of all the resources that go into verification, hardly any of the current proposals for recovery make use of the results of offline verification. Perhaps this is because the results of verification are often only partial, and currently there are no ways to use partial verification results. We believe that the lack of a formal framework for recovery makes it unclear what form of partial results would be useful. Verification is commonly viewed these days as a technique for catching bugs, and how does one make use of bug reports to reduce recovery overhead?

In this section, we take a fresh view of this matter. We contend that results of offline (design) verification, even if partial, can be gainfully used to reduce the overhead of recovery. Rather than view verification simply as a way of catching bugs, we view it as a technique for exploring the reachable and erroneous state spaces of a system. State coverage, even if partial, can be used to reduce the overhead of the recovery strategies proposed in Section 4.

From the viewpoint of reducing recovery overhead in this paper, the goal of offline verification is to reduce the following parameters of the fault model:

1. N_{steps}: The number of steps from the last error move within which any failure can be detected.
2. N_{err}: The number of substitution errors in the guarded command program representing system $\mathcal{M}$.
3. N_{emove}: The maximum length of a sub-sequence of error moves that can be corrected.
4. m: The number of candidate repair moves to be considered at each step

We will focus on how partial verification results obtained from finite-state model checking [12, 13, 39] can be utilized to reduce the values of the above parameters. Furthermore, we will focus on model checking techniques that perform a breadth-first exploration of the state space. The widely used method of symbolic model checking [36] falls in this category. An investigation of what can be gained from other verification techniques is left to future work.

Note that model checking is performed on a *closed system*, obtained as the composition of the system $\mathcal{M}$ with its environment. Thus, we assume for this section that the environment is also finite-state. An iteration of model checking thus includes a step by the system or by the environment or both; this is unlike those in the system runs we have seen in this paper so far, where the system alone steps. The form of the composition (asynchronous or synchronous) affects the form of a run of the combined system. With synchronous composition, both system and environment take a step simultaneously (at each "clock tick"), but with asynchronous composition there could be arbitrary interleaving, in general, except that output actions of the environment are synchronized with input actions of the system, and vice-versa. For simplicity, we will assume synchronous composition for the rest of this section. (Dealing with asynchronous composition will be left to future work.)

Figure 2 shows some kinds of partial model checking results that one might obtain in order to prove that a safety predicate $\neg\mathcal{E}$ is an invariant of $\mathcal{M}$.

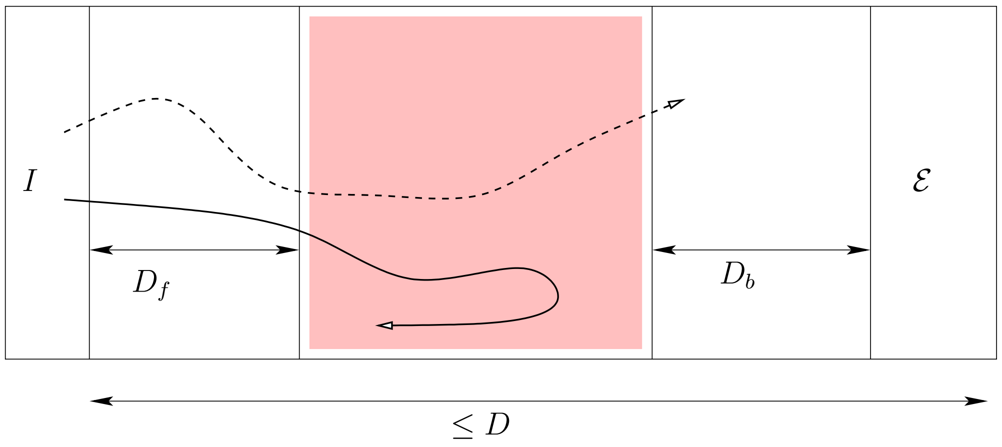

Figure 2: **Partial state space coverage in model checking.** The dashed and solid arrows depict system behaviors.

The most common step in model checking is *reachability analysis*, which computes the set of states that can be reached from some initial state in I. If this set intersects with $\mathcal{E}$, we have found an error state working forward. However, in the event reachability analysis does not complete, we have only partially explored the state space in the forward direction, to a depth of D_f. This is indicated in Figure 2.

Another common form of state space exploration is backwards reachability analysis. Starting with the set of error states, the model checker computes the set of states that can reach $\mathcal{E}$ in $1, 2, 3, \ldots$ steps. If the resulting set of states intersects with the initial states I, then we have found an error trace. However, it is possible that we might only have partially explored the state space working backward to a depth of D_b, also indicated in Figure 2.

Finally, in some situations we may have an upper bound D on the diameter of the state graph, i.e., on the number of steps between an initial state and a reachable error state. The shaded portion of Figure 2 indicates the unexplored part of the state space.

Below, we consider how the above forms of partial state coverage can be used to reduce two of the fault model parameters.

5.1 Reducing N_{steps}

One form of backwards state exploration is to compute the set of states from which an error is inevitable; i.e., if $\mathcal{M}$ continues to follow its transition function from those states, it will enter an error state no matter what sequence of inputs it receives from the environment.

Note that this set of states need not equal $\mathcal{E}$; in fact, the strength of this approach comes from the fact that there could be non-error states from which an error is inevitable because the erroneous action has already been performed and there is a deterministic sequence of inconsequential actions in between it and the error state.

This is performed using the algorithm to compute the set of "uncontrollable" states given by de Alfaro et al. [16]. Conceptually, we iterate the process of computing new uncontrollable states until a fixed point is reached. The iteration is performed as follows:

1. *Initialization:* Set $U_0(s) = \mathcal{E}(s)$.

2. *Iteration:* Compute U_j as

$$U_j(s) = [\forall a \in \mathcal{A}_{inp} \,.\, U_{j-1}(\delta(s,a))] \bigvee [\exists a \in \mathcal{A} \setminus \mathcal{A}_{inp} \,.\, U_{j-1}(\delta(s,a))]$$

   where $\mathcal{A}_{inp}$ are the input actions of $\mathcal{M}$.

   In words, we add a state to U_j if it can reach an uncontrollable state either by all possible input actions or by some non-input action.

Suppose we are unable to complete the above fixed point iteration, but are able to compute U_j for $j = D_b$. In this case, we can detect an error D_b steps before it actually occurs. Thus, we can use an fault model with a new value of N_{steps} which is up to D_b smaller than the original value.

5.2 Reducing m

One way to reduce m is by pre-computing strategies to steer away from error states. This is in fact the problem of synthesizing a controller for safety objectives, for which techniques exist (e.g., see [35]). However, these synthesis algorithms are very expensive, even more than the corresponding verification algorithms.

We can relax this by instead considering a bounded horizon synthesis problem: to pre-compute strategies to steer away from error states reachable in n or fewer steps. This tackles one of the problems with symbolic methods, viz. that they are not very good at searching large state spaces with large diameter of the state graph (i.e. searching "deep"). In particular, it can be hard to use symbolic breadth-first techniques to find out whether a state that can reach an error state within n steps is itself reachable from the initial states.

This problem can be handled as follows. Compute the set of states that can reach an error state in n or fewer steps. Then, for this set, compute a control strategy that avoids the error within $\max(n, N_{steps})$ steps – by allowing an arbitrary action to be taken at one or more points. This can be done by relaxing the transition function of the system. Another approach is to compute all possible ways to avoid error in n steps by computing all witnesses under a relaxed transition relation, using techniques proposed by Sheyner et al [43]. This avoids the need to do a search for alternative repair moves at run-time. However, note that the environment model used offline to find such a bounded horizon control strategy must be conservative.

6 Case Studies

We have performed a few case studies to investigate the topics discussed in earlier sections. Our main example is a network monitor, a packet filtering system to detect malicious traffic, adapted from Varghese's book on Network Algorithmics [45]. We discuss other examples and future work in Section 6.2. All our examples illustrate that when model checking faces computational barriers, online verification and recovery can be effective.

Experimental results reported in this section were obtained on a workstation with 2 dual-core 2.8 GHz Xeon processors and 2 GB of RAM running 32-bit Redhat Enterprise Linux.

6.1 Network Monitor

Varghese describes a simple example of a network monitor (intrusion detection system) in Chapter 1 of his book [45]. The system seeks to drop packets that match a pattern of malicious behavior while forwarding all others. A packet is viewed as a sequence of characters (bytes) of length L. For each possible character c, we have an associated threshold T_c that specifies that c must appear no more than $T_c \cdot L$ times in the packet. In this highly simplified example, a malicious packet is one that violates this threshold restriction for some character.

Varghese works through several implementations of this network monitor, from very simple and slow, to a cleverly optimized version. We chose the most optimized version as the implementation for our experiment, and the second-most optimized version as the specification.[2] We will refer to these as Impl and Spec respectively.

The models we created work as follows. Both Spec and Impl were modeled as synchronous digital circuits that make use of lookup tables (arrays). Thresholds are specified as powers of 2 and stored as shifts. A threshold array `thresh_arr` maintains these shift values. The specification Spec initializes an count array `count_arr` with zeroes, where each entry corresponds to a possible character, and the count is the total number of instances of that character in the current packet. Spec initializes one entry per cycle. Thus, for 256 possible characters, Spec will take 256 cycles to initialize its `count_arr`. After initialization, it proceeds to read and process the bytes in the packet, one byte per cycle. Thus, this process takes L cycles. By calculating the ratio of the count over the threshold on-the-fly, the final pass through the count array to check thresholds is avoided. Thus, the decision on whether to drop or forward the packet is made in $256 + L$ cycles.

[2] Our actual experiments used slightly modified versions, where fields weren't packed into a single word as Varghese describes, but this is orthogonal to the main topic of online verification and recovery.

Even this is a bit slow. The implementation Impl delays initialization at the cost of slightly more storage. Briefly, it initializes only one entry in `count_arr` per round of execution. However, it also maintains a register to keep track of a "generation count", namely the current round number (modulo maximum storable value in the register), as well as generation counts for the last time each character's entry in `count_arr` was updated. If the generation count of the currently read character does not match the global generation count, that character's `count_arr` entry is set to 1 (initialization of 0 plus the first occurrence). By maintaining a large enough generation register it can process each packet in $1 + L$ cycles while avoiding wraparound issues with the global generation count.

For details of the above case study, we refer the reader to the book [45].

In this case study, the specification Spec is a simpler version of Impl that runs much slower. While Spec can be used to detect errors, it cannot be used for recovery as the specification is too slow to run at wire speeds. However, this application is one in which a few failures can be tolerated (a few dropped packets), even as it would be unacceptable to drop all benign packets with a specific pattern. The goal of recovery in this case is to automatically evolve the system towards correctness over several rounds of execution.

We introduced a simple substitution error into Impl. Instead of setting the `count_arr` entry to 1 when a generation count mismatch is detected, we set it to 0. Thus, an off-by-one error is introduced leading to a malicious packet being incorrectly forwarded.

We attempted to model check the composition of Spec and Impl in Cadence SMV [1], but it ran out of 2 GB of memory even for a greatly simplified version. In this version, we set L to 8, considered only 8 possible characters, and maintained a 3-bit generation count register. Thus Spec requires 16 cycles to process a packet, while Impl requires 9 cycles. The combined system operates in rounds of length 18, where Spec uses the last two cycles to check the result obtained by Impl in the first 9 cycles.

On the other hand, an online approach based on the techniques of Section 4 was able to recover the system and pick the correct fix for the fault. We implemented a simulator for the combination of Spec and Impl in C, just as modeled in SMV. If an error is detected by Spec, it enters a recovery mode where it suspends error detection, performs diagnosis, and then performs recovery according to a pre-defined strategy (one of the three described in Section 4). Error diagnosis was performed by re-running a copy of Impl and observing where the counts of the Spec and this copy diverge; this approach provides precise fault localization. Candidate recovery actions included setting the count array entry to a variety of constants (including 0 and 1) as well as increment and decrement operations.

We found that using the C pseudo-random number generator to generate random input characters (packets) quickly led the system into errors. Using a fixed, greedy strategy to recover led to over 50% of the rounds leading to errors. The UR and RWM strategies performed much better, but the number of error rounds in the UR approach showed much greater variance, as expected. In many runs, we observed that the correct repair move was selected after the very first failure.

Our results demonstrate that random testing/simulation is effective at catching the bug as compared to symbolic model checking. When combined with the online recovery methods of Section 4, the system is not only able to detect errors, but also correct them during execution.

6.2 Discussion

We have also performed a second case study. This example is a robot controller modeled in the language of the SMV model checker [36]. It is a real-time system with discrete real-time semantics: a scaled-up version of a case study performed by Campos et al. [9].

The SMV model checker ran on this example for a whole week without returning an answer. Online verification detected an error and successfully recovered from it.

In both case studies, we measured the overhead of recovery only in "logical" terms, i.e., as the number of recovery steps taken over many rounds of execution. We are currently engaged in building the infrastructure to conduct a more quantitative evaluation of the overhead of recovery.

7 Related Work

We organize our survey of related work based on the two main areas of relevance for this paper: online verification and online recovery.

7.1 Online Verification

Online or run-time verification (RV) is the study of techniques to detect design errors at run-time, possibly post-deployment. It differs from traditional testing in that the correctness properties are specified in a formalism such as temporal logic.

Tools and frameworks for run-time verification such as JPaX [26], MaC [28, 29], and LOLA [15] are available. We discuss here a few recent examples of work in RV and refer the reader to a recent survey for details on work published prior to 2004 [18].

Recent references include work by Havelund and Roşu [25] on monitoring safety properties, Elmas et al. [22] for runtime refinement checking of software, D'Angelo et al. [15] for online and offline monitoring of synchronous systems, Sammapun et al. [42] on online monitoring of real-time systems, and Bayazit and Malik [5] for monitoring safety and bounded liveness properties for distributed hardware systems.

Bayazit and Malik discuss one way to integrate online and offline verification. In particular, offline verification of the distributed system is performed using compositional reasoning with the environment assumptions and abstractions that are verified (locally) at run-time. Recovery is only briefly addressed in their paper, and the recovery strategy is designer specified.

7.2 Online Recovery and Robust Computing

Robust computing is the study of run-time techniques to not only detect errors (design or otherwise) at run-time, but also correct for them or at least de-rate the system's performance.

The *recovery-oriented computing* project by Patterson, Fox, et al [37] explores ways to design computers to recover quickly from run-time failures, mainly targeted towards Internet services. An instance of the project is to design reboot-only systems [10], which can handle transient or random error conditions, but cannot deal with deterministic design errors.

Several projects have recently addressed the problem of surviving failures arising from software bugs. Rinard et al. have introduced *failure-oblivious computing* [40], which seeks to execute through buffer overflow problems by returning artificial values for reads that are out of bound. Although potentially unsafe, they demonstrate the utility of being able to execute through an error rather than terminating with an exceptional condition reporting the error. The *reactive immune system* [44] similarly proposed to execute through errors by returning a speculative error code on failure. Qin et al. [38] present Rx, a system for recovering from common bugs in commodity software, such as web servers, such as buffer overruns, races, and memory corruption. The main idea is to change the controllable part of the environment, namely the operating system, by providing safe versions of system and library calls for use during rollback and recovery. However, the approach does not generalize beyond the considered bug classes.

Easwaran et al. [21] discuss *steering*, a technique for predicting failures and taking evasive action in advance, in the context of discrete event systems. Failures are restricted to safety properties and corrective action comprises making error states unreachable. The authors build upon previous work on run-time verification and instead focus on constructing a steerer whose lookahead is more than enough to compensate for communication latency between the steerer and the system. Unlike the proposed work, there is neither use of online learning nor of an integrated approach to verification.

Model-based methods involve the synthesis of fault diagnosis and recovery components from high-level descriptions [17, 47]. An example of a model-based system is Livingstone developed by Williams and Nayak [48]. Their work primarily focuses on device faults (such as in sensors and actuators), as opposed to the design errors that are the subject of this proposal.

Bos and Witteveen [8] have presented a way to optimize a model-based approach for the control of hybrid systems based on pre-compiling the "hard" parts of the constraint satisfaction problem that would need to be solved online. This has a similar flavor to the integrated approach we mention in this paper, but the class of systems and faults addressed are different. Also, we propose to use a broader range of results of offline verification.

Demsky et al. [19, 20] present the concept of *data-structure repair*, which uses a model-based approach to maintaining invariant properties of data structures at run-time in the face of errors introduced by buggy code.

The work on online recovery for hardware systems is more recent. Wagner et al. [46] proposed a technique for recovering from post-silicon bugs in control logic by maintaining a CAM with fixes addressed by "bug patterns". Fault diagnosis is performed offline and the fixes downloaded into the CAM.

7.3 Miscellaneous Topics

Work on *error (fault) localization* is also relevant to this proposal. Zeller's recent book [49] has many references to the literature on this subject, including on error localization for software model checking (e.g., [24]).

While there has been prior work on fault models for design errors, especially in the field of software testing (e.g., [30]), the model we propose and the use for online recovery are quite different from those considered in prior work.

Finally, we mention that work on online verification can benefit from recent work on hardware support for rollback and recovery [27, 50].

8 Conclusions

This paper has proposed an integrated approach to verification to overcome the shortcomings of offline (design) and online (run-time) verification. We have given a formal framework for a large, useful class of finite-state systems, including a fault model for design errors. Strategies for recovering from failures have been explored and the use of learning proposed as a way to optimize recovery overhead. Techniques for leveraging even partial results of offline verification to reduce the cost of online verification and recovery have been presented. A case study showed that online verification and recovery could be effective in a case where (offline) symbolic model checking was unable to complete verification. Our results are a step towards building reactive systems that are robust to failures and self-evolve towards correct systems.

The techniques proposed in this paper can be used even pre-deployment. They can be used in conjunction with dynamic analysis methods, including testing and simulation, in order to assist designers with diagnosing and fixing bugs.

There are many avenues for future work. We are performing further case studies including a comprehensive study of our current fault model. We intend to expand this model in the future and extend our recovery methods to handle a larger class of faults than has been explored herein.

Acknowledgment

The work described in this report has benefited greatly from conversations I've had with many people, including Subramani Arunkumar, Avrim Blum, Ras Bodik, Randy Bryant, Kurt Keutzer, Jim Larus, Edward Lee, Sharad Malik, Jan Rabaey, Shankar Sastry, George Varghese, and Jeannette Wing.

References

[1] Cadence SMV model checker. Available at `http://www.kenmcmil.com/smv.html`.

[2] International technology roadmap for semiconductors: 2004 update: Design. Available at `http://www.itrs.net/Common/2004Update/2004_01_Design.pdf`.

[3] The MicroC/OS-II system. http://www.ucos-ii.com/.

[4] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan.-Mar. 2004.

[5] Ali Alphan Bayazit and Sharad Malik. Complementary use of runtime validation and model checking. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 1052–1059, 2005.

[6] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] Avrim Blum. On-line algorithms in machine learning. In *Online Algorithms, The State of the Art*, volume 1442 of *Lecture Notes in Computer Science*, pages 306–325, 1996.

[8] André Bos and Cees Witteveen. Compilation to speed up the control of hybrid systems. In *Proceedings of the fifth annual conference of the Advanced School for Computing and Imaging (ASCI'99)*, pages 243–250, 1999.

[9] Sérgio Vale Aguiar Campos, Edmund M. Clarke, Wilfredo R. Marrero, and Marius Minea. Verus: A tool for quantitative analysis of finite-state real-time systems. In *Workshop on Languages, Compilers, & Tools for Real-Time Systems*, pages 70–78, 1995.

[10] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot - a technique for cheap recovery. In *6th Symposium on Operating System Design and Implementation (OSDI)*, pages 31–44, 2004.

[11] Edmund Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371, 2003.

[12] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[14] Computing Research Association. Grand research challenges in information systems. Available at `http://www.cra.org/reports/gc.systems.pdf`.

[15] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. LOLA: Runtime monitoring of synchronous systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME)*, pages 166–174, 2005.

[16] Luca de Alfaro, Thomas A. Henzinger, and Freddy Y. C. Mang. Detecting errors before reaching them. In *Proc. Computer-Aided Verification (CAV)*, LNCS 1855, pages 186–201, 2000.

[17] Johan de Kleer and James Kurien. Fundamentals of model-based diagnosis. In *SafeProcess 2003*, 2003.

[18] Nelly Delgado, Ann Q. Gates, and Steve Roach. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering*, 30(12):859–872, December 2004.

[19] Brian Demsky and Martin C. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 78–95, 2003.

[20] Brian Demsky and Martin C. Rinard. Data structure repair using goal-directed reasoning. In *27th International Conference on Software Engineering (ICSE)*, pages 176–185, 2005.

[21] Arvind Easwaran, Sampath Kannan, and Oleg Sokolsky. Steering of discrete event systems: Control theory approach. In *Proc. Workshop on Run-time Verification (RV)*, 2005.

[22] Tayfun Elmas, Serdar Tasiran, and Shaz Qadeer. VYRD: Verifying concurrent programs by runtime refinement-violation detection. In *Proc. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI)*, pages 27–37, 2005.

[23] Yoav Freund and Robert E. Schapire. Game theory, on-line prediction and boosting. In *9th Annual Conference on Computational Learning Theory (COLT)*, pages 325–332, 1996.

[24] Alex Groce, Sagar Chaki, Daniel Kroening, and Ofer Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer*, 8(3):229–247, June 2006.

[25] Klaus Havelund and Grigore Roşu. Efficient monitoring of safety properties. *Software Tools for Technology Transfer*, 6(2):158–173, 2004.

[26] Klaus Havelund and Grigore Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.

[27] Intel Research Pittsburgh. Log-based architectures project. http://www.pittsburgh.intel-research.net/projects/lba.html, 2006.

[28] Moonjoo Kim, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Java-MaC: a run-time assurance tool for Java. In *1st International Workshop on Run-time Verification*, volume 55(2) of *Electronic Notes in Theoretical Computer Science*, 2001.

[29] Moonjoo Kim, Insup Lee, Usa Sammapun, Jangwoo Shin, and Oleg Sokolsky. Monitoring, checking, and steering of real-time systems. In *2nd International Workshop on Run-time Verification*, volume 70(4) of *Electronic Notes in Theoretical Computer Science*, 2002.

[30] D. Richard Kuhn. Fault classes and error detection capability of specification-based testing. *ACM Transactions on Software Engingeering and Methodology*, 8(4):411–424, 1999.

[31] Nancy G. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Publishing Company, 1995.

[32] Nick Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4):285–318, 1987.

[33] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. *Information and Computation*, 108(2):212–261, 1994.

[34] Sharad Malik. A case for runtime validation of hardware. In *Haifa Verification Conference*, pages 30–42, 2005.

[35] Freddy Y.C. Mang. *Games in Open Systems Verification and Synthesis*. PhD thesis, University of California, Berkeley, 2002.

[36] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1992.

[37] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical report, University of California, Berkeley, March 2002.

[38] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating bugs as allergies - a safe method to survive software failures. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 235–248, 2005.

[39] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, number 137 in LNCS, pages 337–351, 1982.

[40] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *6th Symposium on Operating System Design and Implementation (OSDI)*, pages 303–316, 2004.

[41] G. Rothermel and M. J. Harrold. Empirical studies of a safe regression test selection technique. *Software Engineering*, 24(6):410–419, 1999.

[42] Usa Sammapun, Insup Lee, and Oleg Sokolsky. RT-MaC: Runtime monitoring and checking of quantitative and probabilistic properties. In *Proceedings of the 11th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2005.

[43] Oleg Sheyner. *Scenario Graphs and Attack Graphs*. PhD thesis, Carnegie Mellon University, 2005.

[44] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a reactive immune system for software services. In *Proceedings of the USENIX Annual Technical Conference*, 2005.

[45] George Varghese. *Network Algorithmics: An Interdisciplinary Approach to Designing Fast Networked Devices*. Morgan Kaufmann Publishers, 2005.

[46] Ilya Wagner, Valeria Bertacco, and Todd Austin. Shielding against design flaws with field repairable control logic. In *Design Automation Conference (DAC)*, 2006. To appear.

[47] Brian C. Williams, Michel D. Ingham, S. H. Chung, and P. H. Elliott. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE*, 91(1):212–237, 2003.

[48] Brian C. Williams and P. Pandurang Nayak. A model-based approach to reactive self-configuring systems. In *AAAI/IAAI, Vol. 2*, pages 971–978, 1996.

[49] Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan-Kaufmann, 2006.

[50] Pin Zhou, Feng Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iWatcher: Efficient architectural support for software debugging. In *31st International Symposium on Computer Architecture (ISCA)*, pages 224–237, 2004.