Preprint: PRDC 2004: 10th IEEE Pacific Rim International Symposium on Dependable Computing, March 3-5, 2004

Representing User Workarounds As A Component Of System Dependability

Christopher Martin
Bosch Research & Technology Center
Pittsburgh, PA, USA
chris.martin@rtc.bosch.com

Philip Koopman
Carnegie Mellon University
Pittsburgh, PA, USA
koopman@cmu.edu

Abstract

Evaluation of system-level dependability can benefit from representing and assessing the effects of user workarounds as a response to system component failures. We assemble sequence diagrams that represent UML scenarios into mission graphs that contain all possible paths from a particular mission starting point to a particular mission success goal point. Analysis of these graphs reveals potential dependability bottlenecks and the existence of possible workarounds that can be intentionally added to a design, retrofitted to fit an existing design, or discovered as an emergent property of existing system and user behaviors. Simulations of a moderately complex distributed embedded system demonstrate that this approach has potential benefits for representing and improving system-level dependability by including the ability of users to perform simple workarounds to achieve mission objectives.

1. Introduction

While techniques for evaluating system-level reliability based on a combination of component reliabilities are well known, assessing the true operational dependability of a system requires consideration of many factors beyond just individual component failure rates. These factors include software reliability, system configuration problems, operator training, and the existence of gracefully degrading operating modes. For complex systems it is sometimes even difficult to evaluate whether a system is "working" or failed, because failure of some components may leave enough residual system functionality to provided degraded service or even entirely accomplish a subset of possible usage missions. Thus, an important question in building tools to evaluate system dependability is whether it is even possible to represent some of the important factors that make systems dependable on a practical basis.

In this paper, we explore the issue of representing the well known concept of a user "workaround" for evaluating system dependability, especially in the context of complex, multi-user embedded systems. The class of workarounds of interest are "a procedural change to using a computer system intended to compensate for a hardware or component failure" (sense 2 of the definition of workaround from [Koopman03], although our approach might be extended to encompass other classes of workarounds). Workarounds are traditionally created in response to a problem with a deployed system and are often created in an ad hoc fashion. Workarounds can sometimes provide an alternate way to use a system that accomplishes some or all mission objectives, but are not necessarily practical in all situations. In this paper we present one possible way to formalize the representation of a workaround. Additionally, we present a case study showing how this representation can be useful in portraying intended and actual system operation.

The general approach taken is to build upon the Unified Modeling Language (UML) [UML99] use cases, scenarios, and sequence diagrams to create graphs that represent a precise notion of various missions that can be accomplished with a particular system, whether it be fully functional or partially functional. (Other representations, such as Petri Nets, could also be used with appropriate modifications to the approach.) Examples from an elevator system design of moderate complexity demonstrate that straightforward graphical analysis techniques can yield insight into potential dependability bottlenecks. Simulation results show that the complexity of training users to exploit workarounds can be minimal, yet yield significant improvements in practical dependability.

The work presented in this paper builds upon the previous work in [Latronico01], which showed that UML sequence diagrams could be augmented and combined to create a formal language representing all valid uses of a system. In particular, a complex embedded system typically has many different UML use cases that represent different system capabilities. A user desiring to accomplish a mission using the system typically invokes a sequence of different use cases, with the particular use cases invoked depending on the user's starting state, the desired goal state for the mission, the current system internal state, and any user preference for selecting among alternate courses of action. Each use case can in turn be accomplished via activating one or more UML scenarios, with each scenario being a set of steps performed by the user/system combination. Finally, a UML sequence diagram (SD) is a formal graphical representation of a scenario in which each component of a distributed system sends and receives events as well as processes internal state information.

Thus, the starting point for the work presented in this paper is a UML-based design of a complex distributed embedded system that is represented by a set of SDs. Using such a system involves activating a valid sequence of use cases via traversing a set of corresponding SDs that can be represented as either an expression in a formal language or a path through a graph of connected SDs. We explore the concept of workarounds in the form of user-aided arcs within and between SDs.

The subsequent sections of this paper more fully explore the idea of representing humans as a system component for the purposes of evaluating the effects of workarounds on system-level dependability. Section 2 discusses existing work in related fields. Section 3 proposes augmenting UML-based designs to create mission graphs as a formal representation of workarounds. Section 4 presents experimental results from applying the proposed approach to an embedded system design of moderate complexity. Sections 5 and 6 present general observations and conclusions.

2. Related work

In systems that incorporate user actions as part of system operation, we observe that people can compensate for partial system failures via an approach generically known as a "workaround." More precisely, if a system offers multiple mechanisms for an intermediate step toward accomplishing a mission goal, an adequately trained user can try alternate mechanisms if any single mechanism fails due to hardware component failure, design defect, or even user error. Moreover, if multiple paths exist to accomplish a goal, users can choose paths that circumvent partial system failures. We assume that users have adequate knowledge, written instructions, training, or insight into system operation, and that they choose productive actions rather than make matters worse when attempting workarounds. We do not address the issue of mistakes made by users such as performing incorrect steps or omitting steps in a procedure.

While there are many case studies of workarounds, there is little discussion of formalizing the concept of a workaround as part of a system design representation. A thorough definition of the term "workaround" and discussion of related concepts is provided in [Koopman03]. The value of user-available workarounds in desktop, end-user software and in office systems has been explored in [Gasser86]. An embedded system case study of electronic medical equipment suffering from a partial failure due to a Y2K bug was examined in [Manning99]. Some computer-aided design literature considers workarounds to be subversion of a task (e.g., [Day96]), but we consider workarounds in a positive light, and assume that the user is attempting to help a partially failed system fulfill mission requirements.

The area of human-computer interaction (HCI) often focuses on the system interface rather than how a human augments a system's dependability. The GOMS framework [John96], and the related Soar architecture [Rosenbloom93] are largely concerned with the goal of evaluating the performance of the human interface component of a system, rather than that of the underlying human+computer system as a whole. While the role of HCI design is critical to creating robust systems, we seek not just to evaluate HCI, but rather to analyze the functional effects of user actions upon dependability.

Cognitive systems engineering uses a notion of workarounds compatible with ours. [Hoffman02] does not provide an explicit definition for a workaround, but considers it to be a person using knowledge to plan and execute an alternate path to a goal when encountering some malfunction. Task analysis is a general technique to represent tasks as a sequence of steps, and is compatible with our approach. In the area of task analysis, [Albers98] considers workarounds to be activities that people undertake when encountering error conditions or exceptional operating conditions, but argues that goal-driven approaches are necessary rather than a preplanned task analysis. While it might be impractical to foresee all possible failure modes, we think that it is important to catch and represent likely failure modes at design time, or incorporate important failure modes into a design framework in a form that might be considered an off-nominal task analysis.

A paper that presents a similar problem representation is [Poelmans99], which defines a workaround as "a coping strategy that deviates from the strategies that have been defined in a WFS" (Work Flow System), which can be similar to a mission graph. However, Poelmans does not discuss representing a workaround within a WFS as we do.

There are of course many well known techniques for assessing system dependability, although few seem particularly conducive to the exploration of workarounds as a system dependability aspect. For hardware, techniques such as Failure Mode Effects and Criticality Analysis (FMECA) represent the effects of hardware component failures, or subsystem functional failures (*e.g.*, as discussed in [McCollin99]). A FMECA presents an opportunity to document a workaround as a mitigation of a component failure, but does not make workarounds part of the intentional system design. Our work differs in that it examines the effects of deleting a step from a sequence of steps, regardless of root cause, rather having to deduce how a component failure affects attempted operations. Thus it complements a FMECA and other approaches such as Fault Tree Analysis.

[Mo99] uses a hierarchical-tree based approach to representing plans that consist of composed sets of actions, which might be used to deal specifically with workarounds. We have taken a mission-graph based approach instead.

Some previous work has attempted to include people in evaluating the dependability of entire systems. In [Brehm96], system dependability is evaluated based on models constructed from the specified characteristics of each of its components, including software, hardware, and people. However, the underlying inputs to the model (particularly software and human error representations) do not seem to take into account the complex interactions these components can have with the rest of the system, and do not seem to encompass the fact that users can help the system as well as cause failures. [Brown02] acknowledges that human operators can both help and hurt system dependability via diagnosis and maintenance actions.

Similar to the goals of our approach, [Zemany91] aims to provide a mission-level reliability evaluator. That approach takes into account that not all subsystems are required at all phases of a mission, but adheres to the principle of combining component-based failure rates to compute the probability of a mission success. While degraded operating modes are addressed, the specific concept of user workarounds is not discussed.

A subset of preliminary results for this paper appeared in [Latronico01b]. That paper proposed mission graphs as a potentially useful problem formulation, but did not include an assessment of effectiveness and did not discuss looking for single arc/node points of failure as a key approach to evaluating dependability.

3. Mission graphs

Evaluating the system as it performs as a whole, just as if it were deployed in the field and in use by the customer, is essential in making a good reliability estimate [Musa96]. While software testing techniques often use the concept of operational profiling to weight reliability estimates by the frequency with which different system components are activated, we desire a technique that encompasses the notion of mission success hinging on the successful completion of a sequence of tasks rather than the probability of successful completion of a single step in such a sequence. As a result, we evaluate system-level dependability in terms of paths through the set of interconnected SDs that form the valid uses of the system.

3.1 Mission graph dependability

If examined from the user perspective, dependability can be seen as the probability of a user successfully completing a mission, which in the general case consists of a series of tasks. For example, a mission may be riding an elevator, which consists of a series of tasks such as calling the elevator, boarding it, designating a destination floor, and so on. The key to incorporating the effects of workarounds in a system is to create a flexible, but formal and exact, definition of what constitutes a successful mission (*i.e.*, what different sets of system operations can be considered to result in a successful delivery of service).

One way to represent dependability, then, is a graph depicting the possible user paths through the system, which we call a *mission graph*. Building on the work of [Latronico01], the set of valid uses of a system can be described by a graph representing the possible strings created by a formal language representing the system design. Within such a graph, a vertex is a sequence diagram and an arc is a path connecting one SD to a valid successor SD. (The complexity of an SD is not rigorously defined by the UML; however for our purposes we assume that each SD is roughly analogous to a basic block in compiler terms, and thus is for practical purposes a set of system actions that need not be broken up into smaller pieces.) The entirety of the system graph is in general not needed to represent a specific mission. Rather, a subset of the system graph suffices to represent the paths of interest through all possible sequences of system operations.

A *mission graph*, then, is a directed graph consisting of a starting state, a goal state, and all possible intermediate vertices and arcs that lead from the starting state to the goal

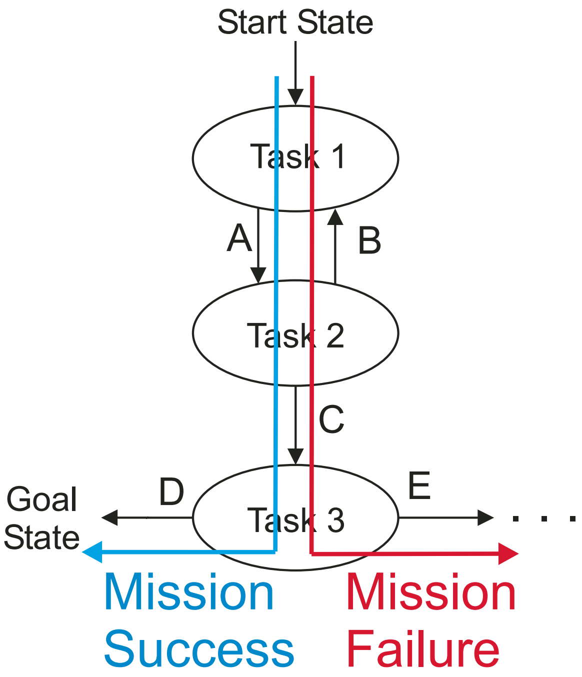


Figure 1: Example User Mission Graph.

state via some valid path through existing SDs. (Additionally, all arcs out of any included graph vertex must be included as described in the next paragraph.) Figure 1 shows that a *mission success* is accomplished by traversing a valid path through the mission graph from start state to goal state. In cases where recursion or iteration are necessary to represent mission successes, it may be desirable to represent mission graphs as symbolically manipulated formal languages rather than directed graphs; however directed graphs are appropriate as the common case and are used in this paper to illustrate the concepts involved. (Limits to recursion/iteration depend on the application. As a practical matter, since system users are involved in determining many arc selections, breaking recursive graph traversals occurs when a user gives up and tries an alternate path or abandons use of the system.) The inclusion of graph cycles is important in representing real systems, because it permits users to revisit states when experiencing emergent or transient system failures. In our work mission graphs have been assembled by hand, but automation seems feasible.

A *mission failure* is a path that does not achieve a desired goal, but instead results in a path through the mission graph that either leads irrevocably away from the intended goal or becomes trapped in a graph cul-de-sac. Mission paths that lead away from a mission success are those that activate an arc from which return to the mission goal is impossible, which corresponds to activating an arc with no destination vertex within the mission graph. (It is possible that some alternate and useful goal state could still be reached in such a situation, but using a mission graph to represent that alternative is beyond the scope of this paper.)

It is important that mission graphs be directed rather than undirected graphs because, in general, visiting a graph vertex produces enduring side effects within the system. As an example, vending money from an automated teller banking machine is an irreversible process. A compensating backward arc can be provided (*e.g.*, a workaround of depositing money in an amount compensating for money withdrawn from a bank), but that is not at all the same as “un-pressing” a button or otherwise intrinsically reversing a side effect via backtracking over a previously executed directed arc.

Component failures can be represented via removing a vertex and/or arc from the mission graph. (In this work we assume a one-to-one or one-to-many mapping from component failures to arc removals.) Because a mission graph is selected from a fully functioning system design, a missing vertex or arc can introduce a number of dead end paths that result in mission failures if not compensated for. Dead ends can be encountered when a user follows a path through the mission path and does not have arcs available for traversing back to a decision point to attempt an alternate path (or, if no alternate path exists).

Of course it is possible to have multiple starting states and multiple goal states concurrently available for use in a system. Creating a distinct mission graph for each possible pair suffices for implementing the methods described herein, but no doubt there are more refined approaches possible that consider multiple sets of starting and goal states in a single multi-mission graph ([Mo99] addresses multiple goals, but uses a different formalism for representing system activity.)

Representing system use as a mission graph not only gives a formal representation of system operation, but also implicitly encompasses the issue of user interactions and training. In the general case, each vertex of a mission graph has multiple exiting arcs. In some cases the exit arc taken depends on system state entirely within the purview of the computer-based system. But in other cases the user determines the arc taken via selecting a button to press or selecting a course of physical action (such as whether to stay on an elevator or exit an elevator when doors are open, based on whether the elevator is at the correct destination floor). At this point we assume that the user has enough system knowledge to make progress toward the system goal if it is possible to do so, and defer discussion of how sophisticated a user must be to make such selections to an example application in a later section.

3.2 Dependability bottleneck detection

A *dependability bottleneck* can be said to exist in a mission graph if there is a single arc that, if severed, would partition the mission graph such that the starting state and goal state were in separate, unconnected subgraphs. In other words, if a mission can only be completed with success by traversing a particular arc somewhere within a graph, that arc will cause all missions to fail if it is severed. By exploiting the duality of vertices and arcs in a graph, a vertex could similarly be a dependability bottleneck if all missions must traverse that vertex. However, we consider arcs as dependability bottlenecks without loss of generality.

Since a specification of “no single point of failure” is common in dependable systems, we look for such single points of failures in evaluating systems. For simple mission graphs, dependability bottlenecks can be found by visual inspection, since what is being sought is a situation in which a single arc points from a portion of the graph on the mission start side to the remainder of the graph on the mission goal side. A failure of this arc would partition the mission graph, making it impossible to complete a mission.

More generally, one could consider *N* concurrent failures in a mission graph. Determining whether a mission graph is vulnerable to *N* concurrent points of failure can be accomplished via applying an s-t min-cut algorithm applied to a directed graph (the “s-t” notation indicates that designated vertices must be on opposite sides of the graph cut, as is the case in this application). If the min-cut is less than degree N, the mission graph would be augmented to fix the vulnerability until a min-cut value of greater than N is achieved. Min-cut algorithms are reasonably efficient and operate in low-order polynomial time. A survey of min-cut algorithms and algorithm performance can be found in [Chekuri97]. (Graph cycles present a problem due to "backward" pointing arcs increasing cut size without providing a path from start to goal. We overcame that by deleting backward arcs before performing graph analysis.)

Once a mission graph has been constructed, it is tempting to assign probabilities to arcs, reliabilities to vertices, and attempt complex computations to derive an overall system availability. While such an approach can certainly be contemplated, it is not at all clear how realistic it would be to obtain the large number of numeric values required for such an exercise. Thus we limit our objectives to one, rather simple, metric that appears to be useful in practice based on identifying single points of failure.

3.3 Dependability bottleneck remediation

Once any dependability bottlenecks have been identified, designers can use a variety of approaches to remediate vulnerabilities discovered. Possible approaches include adding redundancy, adding additional mission paths, and adding additional acceptable goal states.

A time-honored way to increase dependability is to add redundancy. In a mission graph with unweighted arcs this can be represented for analysis purposes as having K redundant arcs on a transition between vertices for K-way redundant hardware.

Additional mission paths can be added by providing workarounds. In the case of a mission graph a workaround is at least one arc, and possibly some vertices, added to provide an alternate path to remediate a dependability bottleneck. Workarounds can be created by adding new capabilities to a design. This is done via adding a new scenario and accompanying SD to a UML-based design and then adding appropriate design and implementation mechanisms to the system.

Workarounds can also be created in many cases by teaching users new procedures that accommodate limitations of existing designs. This can be represented in a UML format by adding a scenario and SD that will work properly given an existing design (*i.e.*, that adds additional constraints and specifications to a design, but does so without necessitating a change in any existing statechart, software, or other existing design and implementation). While this sounds tricky, this is in fact a formal definition of the intuitive notion of a workaround already in common use for software systems – a workaround is simply a way of using a system that accomplishes a goal given a defect or other problem that precludes attaining that goal in "normal" operation. Workarounds are often implemented via instructing a user in a specific procedure devised to be compatible with an existing system implementation.

Obviously there is some added complexity in creating a workaround either via a system design change or a user procedure. Potential issues that go beyond the scope of this paper include instructing users on how to employ workarounds, creating designs that are likely to result in successful workaround attempts by untrained or everyday users, creating designs that are tolerant of erroneous workaround attempts, and so on. Such difficulties aside, it is plainly true that workarounds are a common practice. Thus, it seems worthwhile to formalize their existence and explore their role in system dependability.

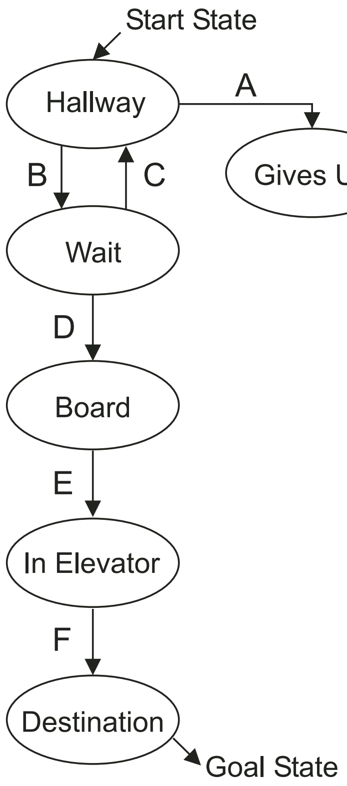
4. Experimental results

Validating the applicability of the mission graph concept and approaches to identifying dependability bottlenecks will of course ultimately require examining many different designs and domains. However, a proof of concept exploration on a design having moderate complexity has provided promising results, and reveals that formalization of even seemingly straightforward system designs can yield beneficial surprises.

4.1 An executable elevator design

An executable model of a UML-based elevator design has been used to explore and demonstrate feasibility of the mission graph concept. This elevator design was not a "toy," but rather a model created as part of an embedded systems capstone design course that embodies many of the complexities of a real-world, high performance elevator based on significant industrial experience. The model is a discrete-event simulation of a single-hoistway elevator according to a 26-page detailed requirements document that is traceable to the approximately 10,000-line Java source code implementation.

Because users of this system play an important role in achieved dependability, the simulation's model of the passengers is thorough and is included as a specific portion of the system requirements and implementation. Passenger behavior encompasses 20 distinct behaviors such as "A Passenger *p* at Floor *f* where a CarLantern(desired_direction(*p*)) is On shall attempt to enter the Car if the Door is sufficiently far open." (The concept of "sufficiently far" is the subject of a different behavioral specification involving the physical size of the passenger, and "attempt to enter" is the subject of specifications involving how multiple passengers queue for exit/entry and respond to a full elevator or a



Arc	Description
A	Passenger times out (frustrated)
B	Passenger presses call button
C	Passenger times out (excessive wait)
D	Doors open / lanterns activate
E	Doors close, elevator travels to destination
F	Passenger exits elevator

Figure 2: Elevator Example.

collision with closing doors.)

The example mission graph in Figure 2 is a simplified mission-level depiction of a user attempting to reach another floor in a building via elevator. The bulk of the example revolves around boarding an elevator (the details of exiting have been omitted in the interest of simplicity, but are included in full in the executable model). The system design and model also include main elevator drive control, acceleration profiles, passenger queuing to account for congestion in loading and unloading, safety interlocks, multiple operating modes, and other similar factors.

The elevator simulation workload represents the conditions a single hoistway elevator in a small office building might be subject to over the course of a typical 24 hour business day. The following test scenario descriptions highlight the different passenger traffic patterns that occur and are generally representative of normal elevator usage:

- LIGHT_TRAFFIC: (extremely early in morning/late at night) When the elevator is occupied, it only contains one or at times two passengers. There is little contention for its use, and the elevator remains more than 30% idle. (The selection of 30% idle as a number is arbitrary, but generally representative.)
- MEDIUM_TRAFFIC: (mid-day use) Moderate contention for the elevator; idle 5% to 10% of the time.
- HEAVY_TRAFFIC: (morning/evening rush hour) Constant contention for elevator; idle 0% of the time with at least one user at some floor waiting for the elevator to arrive at all times.

For each experiment below, long enough simulations were run at each level of traffic intensity to ensure representative results, with each simulating covering at least one hour of simulated passenger loads. However, the precise numeric values of these simulations are unimportant. The important point to note is the effect of differing traffic loads on performance in general and the effect of adding or subtracting arcs from the mission graph on overall system utility (*i.e.*, how well did the elevator deliver passengers with failed or enhanced mission graph arcs?).

4.2. Verifying effects of a dependability bottleneck

In the first experiment, we verified that breaking a single-arc dependability bottleneck in a mission graph actually produced a system failure as expected. Figure 2 shows that the arc labeled D is the only transition from the Wait to Board states. This arc represents the model of how passengers respond to hallway lanterns (the "going up"/"going down" directional arrow lights mounted in the hallway or on the door frame of elevator cars). In particular, passengers only enter the elevator car when the appropriate directional lantern is lit. Table 1 shows that when this arc is broken, representing a component failure such as burned out light bulbs in both directional lanterns, the worst case travel time for a passenger is that they are never delivered. Passengers don't know that the car is headed in an appropriate direction when the car doors are open and thus never enter the elevator.

In this experiment the approach of looking for a single-arc partitioning point in the mission graph did in fact re-

Table 1: Increase in delivery time with failed car lanterns.

Performance Decrease Due to Broken Car Lanterns			
	LIGHT_TRAFFIC	MEDIUM_TRAFFIC	HEAVY_TRAFFIC
Users	N/A[1]	N/A[1]	N/A[1]

[1] Passengers were never successfully delivered in this test

veal this dependability bottleneck that resulted in an inability of users to complete missions. This problem is resolved in real elevators in several ways. One way is the addition of audible chimes in addition to lanterns (typically one chime means “up”; two chimes means “down”), providing a redundant interface that yields a second arc in the mission graph parallel to arc D. A second way is to train passengers to be a little more clever in entering the elevator, such as boarding when the opposite lantern is extinguished rather than the appropriate lantern is lit. This in fact constitutes a workaround (a second arc parallel to D activated via a distinct and less obvious behavioral rule) that improves system dependability, although a more interesting workaround experiment is discussed in a later section. A final way that such a problem is resolved in practice is that people eventually tire of watching the elevator stop at their floor without signaling them to enter, and just enter the elevator without any specific workaround in mind other than compensating for the fact that lanterns must somehow be broken. This final workaround might cause users to enter an elevator traveling in the wrong direction, but has the virtue that no specific user insight into elevator behavior is required and users eventually get delivered to their destination.

Table 2: Increase in delivery time with failed hall call buttons.

Performance Decrease Due to Broken Elevator Summoning Buttons			
	LIGHT_TRAFFIC	MEDIUM_TRAFFIC	HEAVY_TRAFFIC
Users	N/A[1]	178%	27%

[1] Passengers were not successfully delivered in this test after one hour of wait time

4.3. Discovering a false dependability bottleneck

In the process of testing all single-arc bottlenecks, a false bottleneck was discovered. Far from disproving the utility of the approach, this experiment instead illustrated an important point about implicit redundancy that tends to be present in robust everyday systems.

In the example elevator mission graph, transition B is the only arc provided that allows the user to transition out of the initial Hallway state to the Waiting state, which occurs when the user successfully presses the hall call button (this is the appropriate “up” or “down” button in the hallway used to summon the elevator car and provide a hint to the elevator as to intended destination). However, if a hall call button is broken, the elevator is not aware that a passenger is waiting for pick-up, partitioning the elevator mission graph by causing arc B to be inoperative. Thus, breaking this arc via causing a hall call button to fail should result in all passengers attempting to use that button being stranded and failing to complete their missions. (Replication is not addressed in Figure 2. However the executable model maintained separate copies of each hall call button and enabled fault injection into buttons on a selected floor.) Table 2, however, demonstrates that this is not what happened in all cases. In that table performance is relative to worst case passenger delivery time increase compared to a fully operational system under an identical passenger workload (thus, a 0% result means no degradation, and 100% means that the worst case passengers waiting+travel time doubled).

What happened in the medium and heavy traffic cases was that users attempting to press a failed button got lucky when some other passenger exited the elevator car at their floor *and* the elevator happened to be going in the same direction that the user wished to travel in. It so happened that the appropriate hall lantern was illuminated at all stops *whether or not* a hall call had been registered, as is the case with real elevators. It also happened that the implementation of users permitted them to board an elevator without having completed a successful button activation, which is representative of a user arriving at an elevator when the door is already open and bypassing the hall call operation. Given that situation, the heavier the elevator load, the more

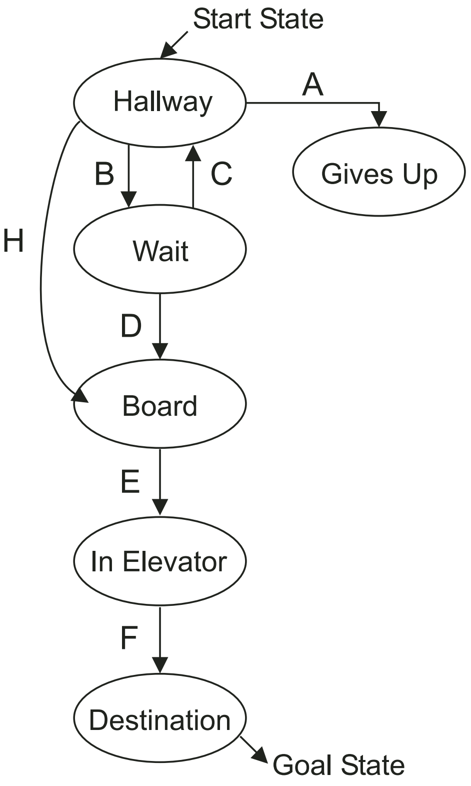


Figure 3: Arc H made explicit for example elevator.

likely it was that some other passenger would choose to disembark at a floor having an otherwise stranded waiting passenger. At very heavy loads the elevator stopped at almost every floor in each direction as a matter of course, resulting in modest performance penalties even with multiple broken hall call buttons. With medium traffic loads it took a while for stranded users to get lucky and board an elevator, and with light loads users had to wait a very long time for an elevator to arrive (in excess of an hour for this particular simulation).

Table 3: Increase in delivery time with hall call workaround.

Performance Decrease Due to Broken Elevator Summoning Buttons			
	LIGHT_TRAFFIC	MEDIUM_TRAFFIC	HEAVY_TRAFFIC
Users	<1%	1%	<1%

While the original design's scenario and SD indicated that passengers press a hall call button before boarding, nothing precluded boarding without pressing a button. The resultant implementation had an implicit new transition between scenarios that permitted passengers to ignore the hall call button if a hall lantern were illuminated, adding an implicit arc H in the mission graph (see Figure 3). It was this implicit arc H that was used by passengers delivered successfully in this experiment, who met the precondition of wanting to board and observing an elevator available even though they hadn't yet pressed a hall call button (which is common in real elevator use as well).

The usual nature of the UML-based use case/scenario design approach is to document typical usage, not represent all possible usages. UML statecharts used for designs must, however, present complete implementations, and therefore are in the general case richer in functionality than the SDs that form their partial specification. From this experiment we see that it is possible (one could argue it is even commonplace in well designed systems) for system designs to have behaviors that may be more robust than indicated by formal design documentation. Thus, the way we look at this experimental result is that identification of a dependability bottleneck prompted a closer look at a critical aspect of a design, and this in turn revealed implicit redundancy that resolved the potential dependability bottleneck in many important situations. It did not completely solve the problem, but adding a mission graph arc to represent the observed system behavior serves to capture the emergent behavior observed and to present a more accurate picture of system dependability. This experiment also illustrates that the effects of concurrent system users are significant, and should be dealt with in evaluating system designs.

4.4. Verifying success of adding a work around

A third experiment verified that adding a user workaround improves system performance, and illustrates the simplicity of some effective workarounds.

The previous experiment showed that passengers could become stranded during periods of light traffic if the hall call buttons they needed were broken. One fairly common way for system designers to deal with such a situation is to assume or foster the existence of a "smarter" passenger that knows about a workaround for such a situation. The workaround desired must offer a redundant arc to arc B, enabling the passenger to transition from the Hallway to the Wait state despite a broken button or broken button acknowledgment light.

Most elevators have two hall call buttons per floor, one in each direction. The workaround added to user behavior consisted of trying to register a hall call in the desired direction, and if that does not succeed attempting a hall call in the *opposite* direction. Figure 4 implies that which button is pressed is at the user's discretion, which is in fact the case in a real elevator system. So, for example, if a passenger presses the "up" hall call button without an accompanying hall button light illumination being observed, the passenger then presses the "down" request button. This is arguably a very simple behavior that would be easy to teach users. Indeed, some impatient elevator users press both buttons as a matter of course. The results on system performance are shown in Table 3, and indicate a negligible performance penalty despite a component failure. For light traffic the elevator dispatching software noted that only one car call button was pressed by the user inside the elevator, and simply went in the correct direction, resulting in no performance penalty in the frequent case that a single user was in the car. In medium traffic, a slight performance penalty was observed because the passenger had to ride the elevator in the wrong direction and upon occasion induced an unnecessary elevator stop if the elevator would have stopped very soon, by chance, at the correct floor going in the correct direction.

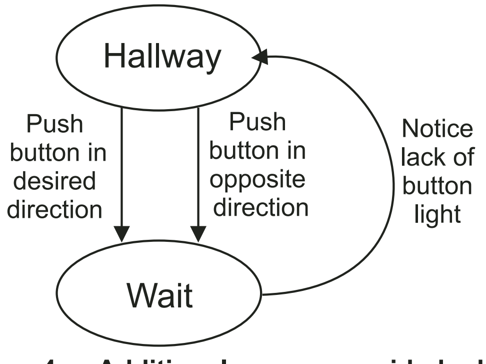

Hallway
Push button in desired direction
Push button in opposite direction
Notice lack of button light
Wait

Figure 4: Additional arc provided by a workaround.

In heavy traffic the elevator stopped at most floors on most journeys up and down the hoistway, so the loss of a single hall call button or spurious hall call button activation made little difference as long as the elevator car did not become completely full.

In this experiment the use of a mission graph revealed a dependability bottleneck, and the addition of a simple workaround was demonstrated to significantly improve system performance in the light traffic case, even given the existence of an implicit, accidental workaround as discussed in the previous section. The medium and heavy traffic cases similarly attained better performance, although they were not broken for practical purposes given the existing implicit workaround. It is worth noting that the particular elevator model built had both "up" and "down" buttons on the top and bottom building floors for simplicity of implementation that provided available redundancy for this workaround. In a real elevator system the dispatcher would have to occasionally register fake hall calls for the top and bottom floors, which have only a single hall call button each, to ensure that no passengers were stranded there. This is an example of a combined user and system behavioral modification to accomplish a workaround. Of course other hybrid schemes are possible.

5. Workarounds as a dependability approach

This paper documents the exploration of representing workarounds in mission graphs for a multi-user embedded system of moderate complexity. Of course not all workarounds will manifest at the mission graph level, since that only represents a particular level of abstraction. Examples of other classes of workarounds include selecting an entirely different mission to substitute for a mission objective that has been foiled, and analog system actions such as jiggling knobs to get them to respond properly. But at the level of mission graphs, our approach should reveal any dependability bottleneck present.

Creating an executable simulation that includes simulated users, rather than a purely paper design, has demonstrated that the techniques can work in a system of this moderate level of complexity. While of course more complex systems would have large mission graphs, such mission graphs are composed of sequence diagrams that would have to be generated anyway in a UML-based design process that uses them. Similarly, a complex system design simulation requires simulated users of some sort. In script-based simulation scenarios, scripts would have to be adapted to incorporate workarounds where necessary.

In the absence of a simulation the concept of a workaround is still viable as an adjunct to FMECA development. Mission graphs focus attention on steps in a usage scenario, and thus examining what happens if such a step fails provides the opportunity for a failure response thought experiment somewhere between a FMECA (which postulates a component failure) and a Fault Tree Analysis (which postulates a mission failure symptom).

Beyond the use of mission graphs, workarounds form a part of everyday life, but have received very little attention in terms of how they fit into system designs. Every system has some limit on the number and types of faults it can tolerate. Once that limit is exceeded, workarounds can form a safety net in any practical application. Of course creating a workaround might require significant ingenuity or might be impossible in some situations. But in instances where workarounds are practical, formalizating their representation and including them as an intentional part of a system design provides a way to document and potentially analyze another dimension of practical system dependability.

6. Conclusions

While the concept of ad hoc workarounds is well known in practice, there is very little work on formalization of their representation for analysis and design. Moreover, workarounds are typically created after system deployment as a stopgap measure when problems are encountered in the field. We advocate explicitly representing the user's ability to work around some failures in the system design phase to ensure that such workarounds are possible, and to document a workaround as a system dependability property that should be preserved when discovered after system deployment.

Mission graphs seem to provide a powerful representational technique for use in comprehensive system dependability evaluation. They can be created by stitching together sequences of scenarios corresponding to a user invoking a set of use cases to accomplish a mission, with most design information required already being part of typical UML-based design processes. Once created, detection of any single-arc dependability bottlenecks provides a way to focus attention on critical aspects of system behavior in order to remediate possible single points of system failure. Use of mission graphs does not necessarily change the underlying ad hoc nature of workarounds themselves; but it does provide a way to formalize the representation of a workaround once it has been created.

Experiments with a multi-user simulation of an embedded system show that using mission graphs to identify dependability bottlenecks for study is feasible for at least three purposes: identifying the location of single-point system failures, documenting previously implicit system-level redundancy that can ameliorate system vulnerabilities, and representing user- or system-provided workarounds to address single point failure vulnerabilities without requiring brute-force redundancy.

While this work proposes that workarounds are worthy of formalization, further work in this area is needed, especially in the area of methodically creating workarounds. Based on these results the following things seem both feasible and promising. Mission graphs can be used to provide a formal representation of single-point system failure vulnerabilities and of the concept of a user "workaround". And, in some cases, workarounds can be quite simple to implement, and need not require complex and highly trained users so long as the user interface and overall system design provides some likelihood users will perform appropriate workaround actions.

7. Acknowledgments

This work was supported by the General Motors Collaborative Laboratory at Carnegie Mellon University and the Pennsylvania Infrastructure Technology Alliance.

8. References

[Brehm96] Brehm, E., "System Dependability Assessment Tool." *Proceedings of the Second IEEE International Conference on Engineering of Complex Computer Systems*, 1996, p. 116-119.

[Brown02] Brown, A., Chung, L. & Patterson, D., "Including the Human Factor in Dependability Benchmarks," *Workshop on Dependability Benchmarking,* 2002, p. F-9–F-14.

[Chekuri97] Chekuri, C., Goldberg, A., Karger, D., Levine, M. & Stein, C. "Experimental study of minimum cut algorithms," *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997, p. 324-333.

[Day96] Day, D., "User responses to constraints in computerized design tools." *ACM SIGSOFT Software Engineering Notes*, vol. 21 no. 5, Sep. 1996, p. 47-50.

[Gasser86] Gasser, L., "The Integration of Computing and Routine Work." *ACM Transactions on Office Information Systems*, Vol. 4, No. 3, July 1986, p. 205-225.

[Hoffman02] Hoffman, R., Klien, G., Laughery, K., "The state of cognitive systems engineering." *IEEE Intelligent Systems*, vol. 17 no. 1, Jan.-Feb. 2002, p. 73-75.

[John96] John, B. E., Kieras, D. E. "Using GOMS for User Interface Design and Evaluation: Which Technique?"*ACM Transactions on Computer-Human Interaction*, Vol. 3, Issue 4, December 1996.

[Koopman03] Koopman, P. & Hoffman, R., "Work-arounds, make-work, and kludges," *IEEE Intelligent Systems*, November/December 2003, p. 70-75.

[Latronico01] Latronico, E., Koopman, P. "Representing Embedded Systems Diagrams as a Formal Language." *Proceedings of the 4th International Conference on UML 2001*, October 2001, p. 302-316.

[Latronico01b] Latronico, E., Martin, C. & Koopman, P., " Analyzing Dependability of Embedded Systems from the User Perspective," *Workshop on Reliability in Embedded Systems* (in conjunction with Symposium on Reliable Distributed Systems/SRDS-2001), October 2001.

[Manning99] Manning, B. R. M. "Technical Guidelines on Embedded Systems." *IEE Seminar on Year 2000: A Practical Approach to Medical Devices and Hospital Systems*, June 1999.

[McCollin99] McCollin, C., "Working around failure." *Manufacturing Engineer*, vol. 78 no. 1, Feb. 1999, p. 37-40.

[Musa96] Musa, J.D., Fuoco, G., Irving, N., Juhlin, B., Kropfl, D., "The Operational Profile," *Handbook of Software Reliability Engineering*, Michael R. Lyu (ed), McGraw-Hill, New York, 1996, p. 167-216.

[Poelmans99] Poelmans, S. "Workarounds and distributed viscosity in a workflow system: a case study." *ACM SIGGROUP Bulletin*, vol. 20 no. 3, Dec. 1999, p. 11-12.

[Rosenbloom93] Rosenbloom, P. S., Laird, J. E., and Newell, A., *The Soar Papers: Research on Integrated Intelligence*, MIT Press, Cambridge, MA, 1993.

[UML99] Unified Modeling Language Specification, Version 1.3, 1999. Available from the Object Management Group. accessed December, 2001.

[Zemany91] Zemany, P. "Applications for Faster to Reliability and Readiness Analysis of Complex Reconfigurable Fault Tolerant Systems." *Proceedings of the 10th IEEE/AIAA Digital Avionics Systems Conference*, October 1991, p. 191-186.