6TH ISRAELI INTERNATIONAL CONFERENCE ON SYSTEMS ENGINEERING
INCOSE IL 2011
Systems Engineering
Look Before You Leap
www.iltam.org/incose_il2011

Mitigating the Risks of Unexpected Events by Systems Engineering

Avi Harel and Menachem P. Weiss
Ergolight Ltd., TechnionIIT, Faculty of Mechanical Engineering
ergolight@gmail.com, mweiss@technion.ac.il

Abstract. Common system engineering methodologies and practices do not mitigate the risks of unexpected events, such as user errors or mode errors, typically attributed to 'force majeure'. The underlying premise is that systems engineers can mitigate such operational risks by considering the human limitations in assuring that the system and its operators are coordinated. The study addressed unexpected events that exist in various kinds of human-operated systems. Because mishaps are typically considered as force majeure, attributed to user errors, the study considered the human limitations in coordinating with the system during the interaction. Prior art demonstrates that usability engineering methods may solve part of the problem. However, in-depth knowledge of the system behavior is required to enable integration of such methods. This implies that the methods for mitigating human risks should be integrated in the system engineering practices. The study extended the scope of system engineering, adding methods and guidelines to help protect from unexpected events. The effectiveness of the methodology was evaluated by case studies, including well-recorded accidents.

INTRODUCTION

Unexpected events are instances of abnormal system behavior, which might be unfortunate. This term complies with the "bad apple" approach (Decker, 2006), by which the mishap is due to the trigger (the "old view"), instead of the situation (the "new view"). Unexpected events are typically attributed to human errors, resulting in productivity loss (Landauer, 1996) and accidents (Sheridan & Nadler, 2006).

This article presents a study about methods for protecting from unexpected events. The study presents a Human Accountability Scale (HAS), enabling to evaluate the accountability-safety tradeoff (Dekker, 2007). The study is motivated by the premise that the system should not enable unexpected events (proactive approach to safety: Peterson, 2001), and if it does, mishaps are inevitable. The study is based on applying the methods on well-known accidents, and examination of the effectiveness of such application.

The framework for protecting from unexpected events is the Systems-Theoretic Accident Model and Processes (STAMP) introduced by Leveson (2004). A key feature of this framework is the association of constrain with normal system behavior. According to this model, accidents are due to the improper setting of constrain, or to insufficient means to enforce them on the system. The methods presented in this study are about setting and enforcing such constrains.

STUDY OVERVIEW

The problem that the study addressed. Common system engineering methodologies and practices do not mitigate the risks of unexpected events, such as user errors or mode errors, typically attributed to 'force majeure'. The underlying premise is that system engineers can mitigate such operational risks by considering the human limitations in assuring that the system and its operators are coordinated.

The industry view of the problem. The study addressed unexpected events that exist in various kinds of human-operated systems, such as:

- Home and personal systems: for example, the ungraceful system response to unintentional changing of the TV channel instead of that of the digital converter
- Safety-critical systems: for example, the crash of transportation systems due to navigating in the wrong mode
- Information systems: for example, unintentional erasing data following the user confirmation of a risky action, which was not accompanied by adequate warning
- Mission-critical control systems: for example, damage due to starting an engine when in the wrong state.

Fulfilling the industry need. Because mishaps are typically considered as force majeure, attributed to user errors, the study focused on human limitations in coordinating with the system during the interaction. Prior art demonstrates that usability engineering methods may solve part of the problem. However, in-depth knowledge of the system behavior is required to enable integration of such methods. This implies that the methods for mitigating human risks should be integrated in the system engineering practices. The chosen direction was by extending the scope of system engineering, adding methods and guidelines to help protect from unexpected events. The methods and guidelines are be based on study of known failures, literature research and interviewing system engineers.

THE EVALUATION FRAMEWORK

The framework for evaluating the risks of unexpected events consists of a classification of unexpected events, and examples of unexpected events that resulted in mishaps. The examples were classified according to the consequences of the unexpected events, in three categories as follows:

- Inter-system mismatch
- Intra-system inconsistency
- Operator's confusion.

Inter-system mismatch. This mishap class is about Systems of Systems (SoS). The examples in this category are of inappropriate communication between systems. Examples:

Friendly fire

- Friendly fire, Afghanistan 2001 (Casey, 2006)
- Friendly fire, Zeelim 1990 (http://www.ynet.co.il/yaan/0,7340,L-1120082-PreYaan,00.html)
- Friendly fire, Zeelim 1992 (http://he.wikipedia.org/wiki/אסון_צאלים_ב).

Unintentional missile launch

The example is of a potential failure due to disregarding the operating scenario in automatic recovery from an intermittent power failure.

Intra-system inconsistency. This mishap class is about state inconsistency between system units. Example:

- Therac 25 (Casey, 1996a)

Operator's confusion. This mishap class is about the mismatch between the system state and the operator's perception of the system state.

- Torrey Canyon, 1967 (Casey, 1996b).
- Flight AF 296 at the 1988 air show (Casey, 1996c)
- Flight IC 605, 1990 (Casey, comments on 1996c).

ANALYSIS OF UNEXPECTED EVENTS

The following model was used to analyze the sources for mishaps such as those in the examples:

Slip management. This document presents a specific pattern of the system behavior, which is typical of mishaps.

In normal operation, all the system units comply with a common scenario. This is the Normal system state. In normal operation, any of the system units can receive various events. The event is considered normal, or expected, if the unit was designed to respond properly to the event. Else, the event is called a slip. Typical slips are due to a hardware faults or to a user's unexpected action.

As a result of the slip, the system state changes to Exceptional. Then, if the system is resilient to the exceptional

event, the system may resume its normal state. Otherwise, a subsequent event may result in a mishap. When this happens, we refer to it as Unexpected.

Exceptional states. Systems are designed to operate according to scenarios. This means that the response of any unit to any event is designed based on an assumption about the operating scenario. In normal operation, all system states assume the same scenario. This scenario is called the system context.

Context compliance. Occasionally, one of the system units may receive an exceptional event (a slip). In response, the operating scenario may change. For example, in case of a unit failure, the operating scenario may change to Unit Replacement. If all the system units operate now according to the new scenario, then the system is context compliant. Otherwise, if not all the system units comply with same context, then the system reaches a state of context inconsistency.

This pattern was applied on the examples, as follows:

Friendly fire in Afghanistan, 2001. The reason for the accident was that the operating state of the GPS slipped out of context: the operating scenario was of Target Acquisition, but the GPS reset following the battery replacement resulted in changing GPS mode to the default Navigation mode. Consequently, the GPS coordinates were set automatically to those of the unit location, which did not comply with the operating scenario.

Friendly fire in Zeelim 1990. The command by the artillery coordinator was a slip, out of the context. The operating scenario was related to the next target, but due to the slip, the command did not comply with the operating scenario.

Friendly fire in Zeelim 1992. While still in the 'no fire' mode, the soldier who launched the missile received a fire command. Based on wrong perception of the exercise stage, he assumed that the operating mode had changed to 'real fire' mode, which was out of context: the operating scenario was still the previous 'no fire' mode.

Unintentional missile launch. The missile reset resulted in the automatic setting to the default Operating mode, which did not comply with the context.

Therac 25. The operator recovered from an inadvertent erroneous setting, but when issuing the command, the system still did not reach the appropriate state. Instead, it was activated when in an unsupported state, which did not comply with the operating scenario.

Torrey Canyon. The operating state of the steering control unintentionally changed to 'Control', which was appropriate for maintenance, but not for navigation. Consequently, the steering control was out of context.

Airbus A320 accidents. Unknown to the pilot, the operating state of the airplane was of "idle/open descent mode", which did not comply with the operating scenario. This mode inconsistency applied to both the AF296 and the IC605 accidents.

Context compatibility. Events are perceived as expected if they comply with the operating scenario, which defines the context of operation. To work properly, all system units, comprising the human operator, should behave according to the procedures defined for this context.

Unexpected events should be expected. Unexpected events are the result of the need to enable the human control over exceptional situations, such as in emergency. At design time, we anticipate main scenarios, but not all possible scenarios. To handle exceptional situations, we need to rely on the human operator. We provide the operator with exceptional control, and we expect the operator to judge when and how to use this exceptional control.

Another main source for unexpected events is design mistakes. It is impractical at design time to consider all possible situations, and to find proper solutions for all circumstances.

This conclusion has also been indicated recently by the need to "prepare for the unprepared" (Paries, 2010).

Reasons for compliance failure. Possible sources of exceptional events include:

- Design that does not fit the operational needs (Airbus 320)
- Interruption of the operational procedure (Afghanistan 2001, Unintentional missile launch)
- Unintentional action that does not comply with the context (Torrey Canyon, Zeelim 1990)
- Failure to comply with a change in the context (Therac 25)

- False perception of the context change (Zeelim 1992)

Enforcing compliance verification. Traditional system design is based on requirement specifications, which are based on requirement analysis, which, in turn, is based on scenario specification. Practically, however, in the specification documents, the requirements are dissociated from the scenarios. The specifications describe features and operational procedures. The problem is that in the documents, the features are not related to the operational scenarios with which they should comply. Also, typical procedure specification documents do not describe the relationships with the system states, and do not describe the desired response to all possible events. Consequently, the derived system design does not incorporate means to match the system activity with the operating scenario.

Common practices of requirement specification do not enforce compliant verification

Extending the specification completeness problem. Typically, the system behavior during the accident development is well understood ad hoc, after the investigation had been completed. However, at design time, the event resulting in the mishap was not expected. A main reason for this may be that it is impractical to consider the enormous number of all possible system states. The ability to solve this problem is critical for ensuring safe operation of complex systems. This problem is an extension of the classical specification completeness problem.

To enable prediction of unexpected events, we need to formalize the operational scenarios, and the relationships between the system state and the operational scenarios.

The validity of specification completeness. Traditional specification completeness is about assuring that the system response to all possible events is well defined, for all possible system states. The completeness problem is about ways to ensure that these events are indeed well defined.

A primary challenge in defining what we mean by 'specification completeness' is to define what we mean by 'well defined'. When applying common practices, this term commonly denotes system resilience. The primary concern then is to make sure that the system might not crash. This definition is insufficient for safety assurance, because it does not consider the human operator needs: to know that the system has received an unexpected event, and that it might be in an exceptional state.

To enable protection from unexpected events, the requirement specification should define the means for ensuring that the human operators can handle the unexpected events.

DEFINITION OF UNEXPECTED EVENTS

Unexpected events may be defined by extending the definition of user errors. This definition is operative, namely, it enables automatic validation that the system state complies with the operating context.

Motivation for formal definition.

Scenario compatibility: To enable automatic validation that the system behavior complies with the context, we need to formalize the rules for state-scenario and for event-scenario compliance. These rules should be presented in the system requirement specifications, and should be considered in the system design.

Context-based specification: The scenario, which formalizes the operational context, should have the following properties:

- It should express the intentions of the stakeholders, and
- It should be relate to the operator's view of the operational stage.

Therefore, scenarios may be described as rules associated with operational procedures.

Assuring specification completeness: To ensure context based specification completeness, for each event, in each system state, for each operational scenario, the specification document should include requirements about:

- Means to notify the human operators about exceptional states
- Means to ensure that the human operators are aware of risky situations
- Means to guide the human operators about the ways to resume normal operation.

An ad-hoc definition of user error: The ad-hoc definition of user errors complies with the observation that a user activity is typically classified as an error if the results are not convenient to the stakeholders (Dekker, 2002; Reason, 1997; also http://www.bmj.com/cgi/content/extract/320/7237/768). This observation means that a user error is not the cause for the misfortune; rather, it is the result of the user activity (Hollnagel, 1993).

Use errors vs. user errors. 'Use error' is a recently introduced term, replacing the popular term 'user error'. The need for changing the term was because of common mal-practice of the stakeholders (the responsible organizations, the authorities, journalists) in cases of accidents (Dekker, 2002): instead of investing in fixing the error-prone design, the management attributed the error to the users.

An operative definition of use error. The ad-hoc definition implies that the use error is the consequence of a user command. This observation might lead to a fatalistic attitude, implying that we cannot avoid use errors. To enable mitigation of the risks of use errors, we need to consider the circumstances of the mishap, regardless of the results. The definition here is intended to enable prevention of such mishaps:

> *A user command is a use error if the results do not comply with the designer's intention.*

This definition may seem fuzzy, as intentions are not in the scope of common engineering practices. To enable detection of unexpected events, we need to rephrase the definition, using engineering terms, such as design requirements and guidelines. An operative definition of a use error is:

> *A user command is a use error if it is not in the scope of predefined user commands appropriate to the operating scenario.*

This definition is operative, because:

- we know what are the predefined commands,
- we can formalize the operational scenarios, and,
- we can assign the user commands to operational procedures or constrains, associated with the operating scenario.

For example, the use error in the Torrey Canyon accident may be described by:

- The predefined commands, including setting the steering control to either of the Manual, Automatic or Control position
- Formalizing the Navigation and the Maintenance operational scenarios
- Assigning the Control position to the Maintenance scenario, but not to the Navigation scenario.

Context-based design. In order to avoid unexpected events, we need to consider the operational context in the design, to apply constrains such that:

- Operational scope: the system is allowed to operate only in predefined scenarios
- Design for consistency: at any time during the operation, the system should operate according to one of the predefined scenarios
- Consistency validation: the system should be provided with means to make sure that all units operate according to the context
- Rule-based consistency assurance: the means may include rules for preventing uncontrolled context change, and for synchronizing the system according to context changes under control.

Formalizing the Unexpected Events. An unexpected event is an exception from the rules, which are typically expressed by operational procedures and constrains. The system design may include:

- An interaction protocol, defining the rules to control the event processing and the state changes, according to the operating scenario
- A scenario tracker, which may hold and update a record of the operating scenario
- An event interpreter, which may verify that the events received comply with the operating scenario

The protocol. In order to enable automatic detection of unexpected events, the system design should include a protocol, which is a model of the normal system behavior, consisting of these rules. The protocol should provide the information about how to respond to each event, in any of the operational scenarios. Based on this protocol, at run time the system should trace the events and update the operating scenario. If in any stage the event does not comply with the protocol, the system may notify about an unexpected event.

A definition of unexpected events. The system engineering extension to the definition of use errors is based on the SoS paradigm, in which the human operator is an intelligent sub system. To extend the definition, we need to extend the scope of the events of user commands.

SoS events. SoS events may be classified as either normal or abnormal. Normal SoS events are those described in the requirement specification, and abnormal events are those that are missing from the specification

documents. By its nature, an abnormal event is unexpected, and the system design should include protection from undesired consequences of such events. However, this work focuses on normal SoS events which might be unexpected.

The designer's view of unexpected events. The designers' view of unexpected events follows:

> *An event is unexpected in an operational scenario if the system was not designed to accept it in the particular scenario.*

An operative definition of unexpected events. The operative definition is:

> *A normal SOS event is unexpected if it is not in the scope of predefined events appropriate to the operating scenario.*

Normal events may be classified as routine or exceptional. Routine events are typically defined in the requirement specifications, and their effect and response are well documented. Exceptional events are sometimes missing, their effect description is often partial, and their response may be missing at all. Both routine and exceptional events are typically expected, but they might also be unexpected, triggering a mishap.

Faults vs. Triggers. Exceptional events are due to faults, and they result in triggers for the mishap.

Faults may be attributed to the hardware, including running out of battery or intermittent power failure. Faults may also be attributed to the user. For example, unintentional key press, mouse click, slip, missing key, double click etc. These are all expected faults. Examples of faults include:

- A system unit is out of service, such as due to corrosion or running out of battery (as in the GPS example)
- Misunderstanding, such as due to communication problems (as in the Zeelim 1992 example)
- Abnormal behavior, such as due to intermittent power failure (as in the unintentional missile launch example)
- User error, such as inadvertent commands (as in the Zeelim 1990 example).

A fault may trigger a mishap. For example, in the GPS accident, the fault was the 'Battery Low' indication. However, this event did not trigger the accident. It was the 'Reset' event, following the battery replacement, which triggered the accident.

Examples. The following table presents the attributes of the definition above to the reference examples:

Mishap	Fault	Trigger	Procedure /constrain	Appropriate for scenarios	Wrong operating scenario
Afghanistan, GPS, 2001	Low battery	Reset	Set default position	Local position setting	Target position setting
Zeelim, 1990	Inadvertent command	Fire command	Exercise plan	Prior position	Current position
Zeelim, 1992	Wrong perception of the exercise stage	Fire command	Launch a missile	Phase II – with live ammunition	Phase I – with no ammunition
Unintentional missile launch	Intermittent power failure	Reset	Set to Operational mode	Operational mission	Exercise
Therac 25	Inadvertent user action	Trigger the beam	Beam spreader plate in place	X ray therapy	Internally inconsistent
Torrey Canyon	Unintentional change of the steering control	Ship heading to the rocks	Steering control in Control position	Maintenance mode	Navigation mode
Airbus A320	Throttle disabled due to a design change	Pilot intends to engage the throttle	Protection envelop	High altitude	Approach airport

DETECTING UNEXPECTED EVENTS

In order to mitigate the risks of unexpected events, we need to detect them. The formal definition of unexpected events presented above is operative, namely, it may be used for automatic classification of events as expected or

unexpected.

Architectures. Three architectures are considered here:

- A single system
- A centralized SoS
- A distributed SoS.

A single system. The Torrey Canyon supertanker is an example of a single system. In the simple form, demonstrating the detection of unexpected events, there are only two operational scenarios: Maintenance and Navigation. The following table presents the protocol:

Event	Condition	System response
Steering control set to Manual	-	Rudder connected to steering wheel
Steering control set to Automatic	-	Rudder disconnected from steering wheel, obeys a predefined procedure
Steering control set to Control	Maintenance mode	Rudder disconnected from steering wheel,
	Navigation mode	Unexpected !!!
Change in steering wheel position	Steering control in Manual position	Rudder follows steering wheel
	else	No effect on rudder

A centralized SoS. There is only one system that traces the scenarios and verifies that the events follow the rules.

Example: Zeelim 1990. The protocol is the plan table describing how many shells should be fired on each stage, in response to which command. The artillery battery is in charge of tracing the scenarios and verifying that they comply with the plan. If an event does not comply with the plan, as was the case in the Zeelim accident, then the artillery unit may hold the fire and notify the forward officer about the deviation from the plan.

A distributed SoS. The system may consist of several systems. The system owns a set of protocols defining the rules for communicating with the other systems. Each system traces the operating scenario, and verifies compliance with that scenario. This configuration may be adequate for multi-control systems, such as in nuclear power station, military support systems, and more.

Example: The GPS in Afghanistan. The system consisted of two sub systems: a GPS used for target acquisition, and an airplane which should launch the ammunition towards the target.

To enable detection of unexpected events, each of the two systems could trace the operating scenario. While communicating, the two systems could notify each other about the operating scenario. Both systems could check that the events comply with the protocol. In case of mismatch between the scenarios, each of them could notify the other.

Design for risk detection. In the Torrey Canyon example, consider the event of "Steering control set to Control" when the supertanker is in Navigation mode. The required system response, according to the response table above, is 'Unexpected'. When designing for risk detection, the designers need to specify how the system should behave when the system response is unexpected. The specification should include the following details:

- Means to avoid escalation
- Method for notifying the operators about risky situation
- Procedure for recovering from the undesired state
- Means to report about this event to the stakeholders.

The stakeholders should be aware that the unexpected occurred, so that in an upgraded version, this event is expected. The system design should include means to record unexpected events, together with the history of events and state changes that ended up with the unexpected event. The system design should also include a default behavior, such as automatic shutdown. And the system design should include means to notify the operators about the unexpected event.

Incremental protection development. The response table for the Torrey Canyon above is limited to the states and events that were involved in the interaction just preceding the accident. In practice, the number of states and events that should be considered is enormous, and it is inconceivable that they all of them may be considered in depth at design time. This means that even when we try hard to list and provide defense against all unexpected

event, we should still assume that some of them might sneak in at run time.

In most cases, the rate of incidents (unexpected events) is much higher than that of actual accidents. In such cases, we can apply the strategy of incremental protection development. To defend against the events not included in the table, we need to provide a default protection, and we need to design for unexpected events as described above. The protection may be added incrementally: we run the system until we encounter an unexpected event (which, hopefully, does not result in an accident). Then we design a change for the particular event, and finally we upgrade the system and reiterate the procedure.

AVOIDING UNEXPECTED EVENTS

The best strategy for mitigating the risks of unexpected events is by avoiding them.

The risk: unintentional control activation. If the system design does not protect from careless control activation, then we should assume that eventually an operator might activate this control. For example, if an On-Off button is not protected from unintentional actuation, we should assume that this will eventually happen.

Controls that have an immediate irreversible effect on the system behavior should be protected from accidental actuation. The protection may be by restricting the access to the control:

- By hardware, namely using a special cover, or
- By software, by prompting the operator to confirm the risky function.

The risk: applying the proper control in the wrong mode. This risk is typical of functional overload of controls. System designers are tempted to save controls and panel space by overloading similar functions on the same control. The operator can actuate the proper function by selection of the appropriate mode. Moreover, certain software development methodologies, such as state-chart and use case specification, and object-oriented design, encourage mode dependent response specification and design. The problem is that the operators do not always keep track of the mode changes, and eventually they operate the mode-dependent controls in the wrong mode. The example of unintentional missile launch demonstrates the risks of mode dependent functions.

Whenever possible, controls should be allocated to single functions. If a control should be not be active certain modes, then safety means should be designed to inform the operator about attempting to activate the control in a wrong mode.

When several functions are assigned to the same control, then safety means should be designed to prevent activation of the control in an unintended mode. The safety means may include:

- Control modulation: assigning controls to objects (in an object-oriented interaction design) or to scenarios (in a scenario-based interaction design), such that each control is assigned with a single function.
- Event threading: a technique of event duplication, such that each copy of the event actuate a single function.
- Warning the operators about the activation of risky functions
- Feedback to the operator about the function that was actually activated.

The risk: incomplete specification. This risk is typical of complex systems. The number of system states is an exponential function of the number of states. Therefore, if the system behavior depends on numerous states, then the number of possible states is enormous, and it is impractical to specify the system behavior in all possible states. Subsequently, if the system behavior is not specified for a particular state, then system behavior when in this particular state is unpredictable. An example of surprise due to incomplete specification is that of the Therac-25 accidents.

Incomplete specification is also a problem of very simple designs, such as that of the Torrey Canyon accident.

The system specification should include a description of default behavior. The default behavior should be the safest for the particular system. The description may include a method for recording the event details, displaying information about the event to the stakeholders, and a common recovery procedure.

The decision about the safest default behavior is challenging. For example, Leveson (2004) has reported on an accident in a chemical processing plant due to freezing the system in response to an exceptional situation.

The risk: automatic problem solving. Automation often results in the system taking the control from the operators. Sometimes, this results in an accident, such as that of AF 296. Analysis of these accidents reveals that they are typical to situations of problem solving, in which the operators are not aware of the problem. This was the case of AF 296 and of three aircraft accidents, reported by Norman (1990).

The system may help with the problem solving by automation, as long as it provides feedback about it to the human operator.

The risk: state mismatch. In many incidents and accidents, the system entered an unstable state prior to the unexpected event. This was the case of the Therac 25 accidents, and many cases of friendly fire.

An unstable system state is a term used to describe a situation of state mismatch between two or more system units. This means that a system unit, which should be 'aware' of the state another system unit, 'perceives' it wrongly. This kind of mismatch is most common in a generalized system, when the human operator is considered as an intelligent system unit. The human operator might misconceive the system state due to various reasons, such as lack of feedback, attention focus on other tasks, vigilance problem etc.

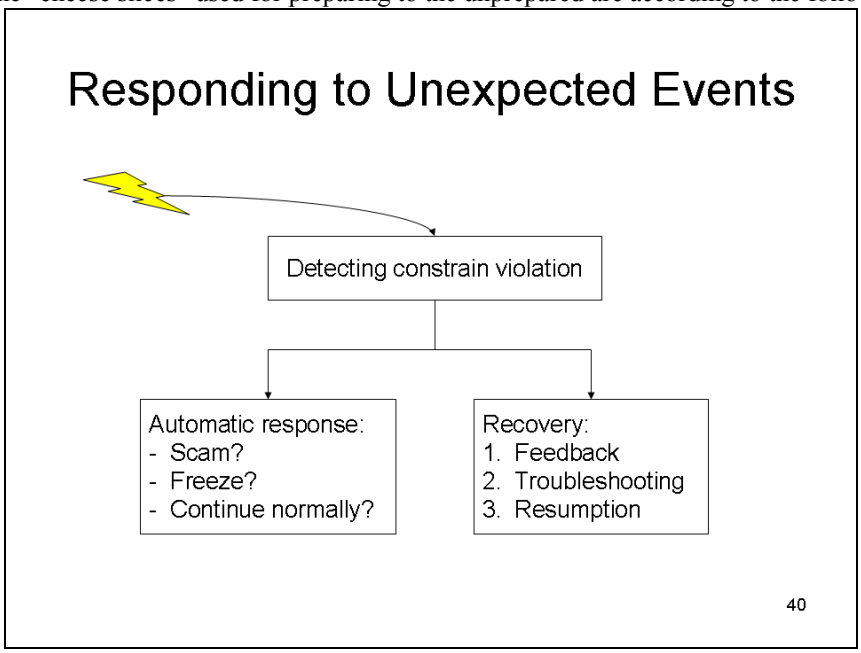
The design should include means to avoid state mismatch. The means may include:

- State modularity: only one unit manages a system states; all the others may inquire the state when they need it. For example, the system may prompt the operator to specify their intention when this is required for the operation.
- State synchronization: On a change in any of the system states, the unit that changes the state may broadcast to the other units about the change, and the other units should confirm the update, to ensure synchronization.

STAGE 6 – RESPONDING TO UNEXPECTED EVENTS

We cannot totally avoid unexpected events, and we always need to "prepare for the unprepared" (Paries, 2010). This document provides guidelines for reducing the damage of unexpected events that we failed to avoid.

Overview. The method proposed for responding to unexpected events is based on the Swiss Cheese model by Reason (1997). According to this model, we need to identify the "safe holes" and to add protection means for each of them. The "cheese slices" used for preparing to the unprepared are according to the following chart:



Detecting Constrain Violation. Following the STAMP model (Leveson, 2004), unexpected events involve constrain violation. The constraints describing normal system operation may be defined according to the document of milestone 2, by operational scenarios. Methods for detecting constrain violation are described in the document of milestone 3.

Automatic Response. A key dilemma in system design is about the way that the system should respond to unexpected event. Examples of response strategies are:

- Scam: brute-force stop of the main activity, as in nuclear power plants.
- Freeze: the objective is to enable reliable investigation of the sources for the unexpected event
- Continue normally: this strategy is the only option for many real-time systems, such as aircrafts.

Any decision about this dilemma is a kind of gamble, because the events are unexpected, and we cannot know

the consequences in advance. For example, Leveson (2004) reported on an explosion in a chemical plant due to freezing in a transient state.

Recovery. Recovery from the unexpected event is a three stage procedure:

- Feedback: the system notifies the operator about the event
- Troubleshooting: the operator finds the source for the exceptional state
- Resumption: the operator enables resumption of the normal operation.

Feedback design. The feedback design should take care of the following subjects:

- Visibility and audibility: make sure that the operators notice the feedback, in all possible operational conditions
- Low rate of false alarms: tune the level of false alarms such that the operators respond to them properly
- Sense of emergency: set the alarms in a way that they alert on immediate risks, and are informative on potential risks.

Troubleshooting design. An example of the troubleshooting problem is the Three Miles Island (TMI) nuclear power accident in 1979. The system provided too many warnings, but the operators failed to identify the source for these warnings.

The challenge of troubleshooting design is to provide a single, exact directive to the source of the exceptional situation. If safety is not compromised, this means adding detectors about all possible malfunctions of all system components. Also, because the detectors can fail, the system should continuously test that they are still functional, and report about their failures.

Many possible faults, such as gas leakage, cannot be detected directly. Troubleshooting such faults may be accomplished by trend analysis, based on measurements of process properties, such as temperature and pressure, and by comparison to simulator results.

Resumption. The resumption of the system operation may be automatic or manual. If the resumption is automatic, the system should provide salient indication about it, and the operator's confirmation may be required, to ensure that the operator is aware of the resumption.

The system specification should include a definition of the starting point for the resumption.

ESTING WITH USERS

Due to the complexity of operational situations, it is not realistic to assume that we can identify all unexpected events during alpha testing. A more realistic approach is to verify at the alpha stage that the system is tolerant to unexpected events, and that the detection of the majority of unexpected events will be postponed to the field testing stage.

The Testing Goal. The testing goal is to identify the unexpected events as early in the development cycle as possible, in order to refine the system specifications, to handle exceptional situations.

The Test Plan. The model for test generation may have two parts:

- Manually crafted tests, for manually validating the software infrastructure for capturing and managing exceptional events,
- Random computer-generated test cases for representatives of situations and event classes.

A special flag may enable automatic recording of exceptional events at test time, instead of stopping the system operation, which is the proper response after the system deployment.

Test-Case Generation. The preferred method of test-case generation depends on the kind of testing. For user testing, the test cases are derived from the user tasks, based on task analysis. For system testing, test case should be generated automatically, for all possible events, applied in all possible situations.

Test-Case Complexity. Brute-force test-case generation is of exponential complexity. In order to find all unexpected events, we need to test the effect of all possible events in all possible situations. The test-case complexity (the number of test cases) is due to the huge number of possible situations, which is the number of specified scenarios multiplied by the product of the number of states that may exist in all state modules:

$$NuOfTestCases \cong NuOfScenarios \times \prod_{m=1}^{NuModules} NuOfStatesInModule(m) \qquad (1)$$

Fortunately, when using the inter-module state transition model, the test case complexity reduces drastically to:

$$NuOfTestCases \cong \sum_{s=1}^{NuOfScenarios} NuOfTransitions(s) \qquad (2)$$

However, the number of test cases required is still too large to craft them manually.

Evaluation. An unexpected event may be due to a bug or a design mistake. If the event is of an expected, yet exceptional situation, then it should be included in the procedure of a scenario describing the exceptional situation. For example, if due to a design mistake the specifications do not describe a scenario for managing intermittent power failures of a system component, then the unit reset after recovery would result in an exceptional event. The fact that the event is classified as unexpected implies that the specification, design and code should be checked, in order to find out the reasons for the wrong classification. In the example, this would imply the need to add a scenario.

Test Management. Some of the exceptional events, specifically the user generated events, require the user confirmation in order to decide about a design change. This implies that the test should stop at each exceptional event. To enable uninterrupted testing, it would be desired that the test program decides on behalf of the user. The options are to either confirm the exceptional event, or to stop for evaluation. Automatic confirmation, recorded in a log file, enables freeing the tester from the tedious confirmation of repeating events. The disadvantage of this method is in case of an unexpected event due to a design mistake or a bug, in which the system might crash.

Post-release Testing. Post-release (field) testing may be based on automatic recording of the user activity on a log file at the customer site, and analysis at the tester site. The analysis employs special indicators of exceptional events, based on anecdotal models, and on special statistics for comparing the exceptional situations with normal situations.

REFERENCES

Bainbridge, L. (1987). Ironies of automation; in: *New technology and human error*, (ed. J. Rasmussen, K. Duncan, & J. Leplat). New York: Wiley

Casey, S. (1996a). Set Phasers on Stun; in S. Casey: *Set Phasers on Stun*, *And Other True Tales of Design, Technology and Human Error*, Aegean Publishing.

Casey, S. (1996b). A Memento of Your Service; in S. Casey: *Set Phasers on Stun*, *And Other True Tales of Design, Technology and Human Error*, Aegean Publishing.

Casey, S. (1996c). A Leap of Faith; in S. Casey: *Set Phasers on Stun*, *And Other True Tales of Design, Technology and Human Error*, Aegean Publishing.

Casey, S. (2006). Death on Call; in S. Casey: *The Atomic Chef*, *And Other True Tales of Design, Technology and Human Error*, Aegean Publishing.

Dekker, S. (2002). *The re invention of human error*; Technical Report 2002-01, Lund University School of Aviation, Sweden.

Dekker, S. (2006). *The Field Guide to Understanding Human Error*, Ashgate.

Dekker, S. (2007). *Just Culture : Balancing Safety and Accountability*, Ashgate.

Hollnagel, E. (1993). *Human reliability analysis: Context and control*. Academic Press.

Landauer, T.K. (1996). *The trouble with computers: usefulness, usability and productivity*. MIT Press.

Leveson, N.G. (2004). A New Accident Model for Engineering Safer Systems, *Safety Science*, Vol. 42, No. 4, pp. 237-270.

Norman, D.A. (1980). Why people make mistakes. *Reader's Digest, 117*, 103-106.

Norman, D.A. (1990). The "problem" with automation: Inappropriate feedback and interaction, not "over automation". In *Human Factors in Hazardous Situations*, D. E. Broadbent, J. Reason, and A. Baddeley, Eds. Clarendon Press, New York, NY, 137-145.

Paries, J., (2010). Resilience and the ability to respond; in: E. Hollnagel, J. Paries, D.D. Woods and J. Wreathall: *Resilience Engineering in Practice*: Ashgate.

Peterson, D. (2001). *Safety Management: A Human Approach*. American Society of Safety Engineers; 3 edition

Reason, J. (1997). *Managing the Risks of Organizational Accidents*. Aldershot, England: Ashgate.

Sheridan, T.B. and Nadler, E.B. (2006). A Review of Human-Automation Interaction Failures and Lessons Learned, NASA.

BIOGRAPHY

Avi Harel

Avi Harel received his B.Sc. (1970) and M.Sc. (1972) degrees in mathematics from the Technion. His work experience includes Rafael and Ergolight, a startup that he initiated. For Ergolight, he developed several software tools, used by system engineers to identify design mistakes resulting in use errors. He is an expert in systems engineering and in usability engineering. Recently, he specialized in the theory and practice of protecting from unexpected events, and specifically from human errors.

Menachem P. Weiss

Menachem P. Weiss received his ME degrees in the Technion, BSc in 1960, MSc in 1968 and DSc in 1973. Res, ass, in UC Berkeley in 1974-6, and later research fellow in Lawrence Berkeley labs, and in the University College, London in 1990/1. Worked for Rafael from 1963 till 1990, as a design engineer, Head of a main Mech. Design group, and Head of a major multidiscipline project, In 1975 received the Israel Price for Defense, by the president of Israel. Later served in the company central management as Director of Procurement. In years 1994 till 2006 served as a Prof. in the Technion IIT, Faculty of Mechanical Engineering. Specialized in developing Systematic Design Methods, Fatigue models and design of materials handling equipment. Recently works in the design of Medical Devices.