The Safety Requirements Engineering Dilemma

Daniel M. Berry
Computer Science, Technion, Haifa 32000, Israel (dberry@cs.technion.ac.il)

Abstract

A key idea followed in the software and system safety community is that an identified hazard is best dealt with by changing the requirements of the system so that the hazard does not even occur. This modus operandi creates a serious dilemma. The hazard identification, that is needed in order to know what hazards to avoid, is best done after the code has been written, because only then are the potential effects of any particular stimulus, event, etc. deducible. However, if the response to the identified hazard is to change the requirements, then this requirements change will happen only after the code is written. Such changes are both expensive and dangerous. So, a means to identify all hazards at requirements analysis time is needed.

Safety

As stated by Modugno *et al*, "A *safe* system is one that is free from accidents or unacceptable losses." [5] An *accident* is an undesired and unplanned, but not necessarily unexpected, event that results in, at least, a specified level of loss [3]. A *loss* is damage to or destruction of property or injury to or death of a living being, particularly a human being. Any system operates in an *environment*. A *hazard* in a system is a state or condition of the system that can, in the presence of a stimulus from the environment, lead to an accident or loss.

Hazard Identification

A key part of a demonstration of a system to be safe is the identification of hazards and an analysis of what to do about the hazard. Choices for a hazard are to 1) eliminate it, i.e., make it impossible, 2) reduce the likelihood of its occurrence, or 3) mitigate its effects. In some cases, the cause of a hazard can be identified, and then it can be controlled or even eliminated.

As with other kinds of errors in a system, it is impossible to predict in advance of deployment and use of the system, all possible causes of all possible hazards. Thus, it is necessary to gather as much information as possible about the behavior of the system under construction, and to hope that this information allows detection of the causes of all hazards before they lead to accidents.

A similar situation exists with system security. Indeed, a system is said to be *secure* if it is free of unacceptable security breaches, i.e., leakage of sensitive information to inappropriate recipients, or destruction or change of information by unauthorized updaters. The difficulty of identifying security *threats* is the same as that of identifying safety hazards. There is a whole repertoire of techniques for identifying and analyzing security threats, and these are very similar in flavor to the techniques used for identifying and analyzing safety hazards.

Hazard Elimination

The preferred approach to dealing with an identified safety hazard is to change the system's requirements so that the hazard is entirely avoided, i.e., to eliminate the hazard. For example, if inputting a non-negative value greater than or equal to 100 will cause a tank to explode, then better than having the system accept any input and check that it is greater than or equal to zero and less than 100, delivering an error message if it is not, is having a user interface that precludes any value outside the legitimate range from being input in the first place, e.g., with a slider bounded by zero at the bottom and 99 at the top, or with an input window that is only two digits wide.

In the security domain also, changing requirements to eliminate security threats rather than checking for errors at run time is the preferred approach.

Hazard elimination requires that hazards be identified as early as possible, i.e., during requirements analysis so that the requirements that avoid specific hazards are known by the time the requirements are being specified.

Hazard Identification Techniques

When one reads the literature on software based system safety, e.g., Leveson's book [3] and other papers [5, 4], he or she is struck by the level of detail of the

models that must be used to do the hazard analysis. Even though these models are called blackbox models because they do not show so-called implementation details, they do show for each stimulus from the environment thought to be relevant, a transition in the state model from any state. This state model is intended to capture the user's mental model of the external behavior of the system [4].

Note that only the stimuli *thought* to be relevant are handled by the blackbox model. One reason for modifying the model is the discovery of another stimulus or another group of simultaneous stimuli, possibly previously known, that is relevant to the safety of the system.

Such detail is necessary to be able to carry out any useful state machine hazard analysis (SMHA), whether by forward search from possible initial states and stimuli or by backward search from known hazards [5].

The software-based system safety community has come to regard such blackbox models as requirement level models simply because it has no choice. Without this level of detail, the hazard conditions are simply invisible, having been abstracted away into states and transitions in which the conditions and sequences of events that lead to accidents are not expressible. In a normal non-safety-critical system, such a model would be called a high-level design or simply a design. Put in terms of Leveson's intent specifications [4], most would consider as requirements only Section 1, *System Purpose*; most would consider as design Sections 2, *Design Principles*, 3, *Blackbox Behavior*, and 4, *Physical and Logical Function*; and most would consider as implementation documentation Section 5, *Physical Realization*.

Moreover, when SMHA fails to identify a hazard that rears its ugly head later, it can be that the underlying blackbox model is just not detailed enough to expose the hazard condition. Indeed, as deployed systems are discovered to present hazards that were not anticipated, a revision of the blackbox model is in order.

The same observation holds for other hazard identification techniques including forward simulation, HAZards OPerability analysis (HAZOP), or deviation analysis, and others described in detail in [3].

Dilemma

Hazard elimination is best done early in the lifecycle, early enough to affect the requirements specification. The hazard identification that is needed in order to know what hazards to avoid is best done after the code has been written, because only then are the potential effects of any particular stimulus, event, etc. deducible. However, if the response to the identified hazard is to change the requirements, then these requirements change will happen only after the code is written. Such changes are both expensive and dangerous. They are expensive because they mean throwing out previously written code, and they are dangerous because attempting to change only the affected code runs the risk of unidentified ripple effects and flakier software. Neither is very palatable. Therefore, the need is for a means to identify all hazards at requirements analysis time.

One Approach Out of Dilemma

van Lamsweerde and Letier (vLL) [7] identify the dilemma and try to identify unexpected agent behavior at specification time and at the goal level without having to wait until design or implementation time and without having to delve into design and code details.

Once a goal-driven requirements elaboration has been carried out to yield a formal specification in the KAOS language, they use formal methods to identify obstacles to requirements satisfaction from the specifications of goals and domain properties and to modify goals, requirements, and assumptions to overcome or mitigate the identified obstacles. It should be clear that this terminology captures exactly the desired way of dealing with safety problems described above.

Obstacle identification itself consists in finding some assertion for each goal and assumption, i.e., an obstacle, that may prevent their satisfaction, verifying that the candidate obstacle is consistent with the domain theory, and determining if this candidate obstacle is satisfiable by trying to find a feasible negating scenario.

A goal can be categorized by the type of requirements it derives from the the agents involved. For each such goal category, specific obstacle categories may be defined, e.g., starvation obstacles for satisfaction goals, hazard obstacles for safety goals, misinformation and forgetting obstacles for informing goals, threat obstacles for security goals, etc. Knowledge of the category of a goal may drive a search for obstacles in the corresponding obstacle category, Thus, vLL too have observed the similarity of methods for dealing with safety and security, and with other problems as well.

The key idea of their paper is that obstacle identification needs to be done as early as possible in the system lifecycle, and as early as possible during requirements analysis, that is, at the time that goals are being identified. The earlier such identification is done, the more freedom is obtained in dealing with the obstacles, at best by changing goals so that the obstacle does not even happen. Clearly, goals here correspond to Leveson *et al*'s intents, obstacles correspond to hazards, and derived objects and operations correspond to the blackbox model. Indeed, vLL confirm this correspondence when they suggest that goals might provide a precise entry point for

analysis, e.g., constructing the fault tree starting from negated goals. Clearly, the steps to formally derive obstacles from goals are a formal realization of blackbox modeling and hazard analysis, which are normally done manually, albeit systematically, and with tools when applicable. In any cases, there is no escaping deriving more details than are normally considered requirements.

Another Approach Out of Dilemma

Anderson, de Lemos, and Saeed (AdLS) observe that a major problem facing developers of safety-critical software-based systems is the major rework of designs and implementations caused by discovery of safety problems too late in the system's lifecycle [6, 2, 1]. Their solution to this problem is to address the system-context and failure-behavior safety concerns during the requirements phase of the lifecycle. The requirements phases is the most efficient time to do so, to insure that safety problems do not survive through to deployment.

The AdLS method for arriving at requirements for safety-critical systems consists in modeling the system and its environment, iterative simultaneous requirements and safety analyses, and documentation of linkages identified between artifacts produced by these analyses with a Safety Specification Graph (SSG). In any iteration, analysis of the requirements specifications yields safety specifications. The analysis of these safety specifications assesses the risks associated with them. If the risks are unacceptable, the safety specifications are modified, leading to restrictions on the requirements specifications. If any of these restrictions compromise the mission, the requirements specifications must be changed also; the cycle continues to convergence. These analyses are both qualitative, to identify the hazards, and quantitative, to calculate the probabilities and thus the risks, and these are basically the same analyses as described above, but adapted to the specific notations suggested by AdLS.

The feasibility of the AdLS approach has been demonstrated by a good number of case studies in the domains of railway systems, chemical batch processing, and avionics systems.

Workshop Case Study

The meeting scheduler case study at first glance seems not to be subject to the ideas of this position paper. Indeed, there appear to be no safety and security ramifications of the problem as it is described in the web page,

http://www.eecs.uic.edu/ buy/case.html .

However, this may even be an example of the problem. It might be that when a concrete design is given, one can see some possible security threats, e.g., if the meeting scheduler would have access to users' files to update their private calendars.

Conclusions

To conclude, it might be that the only solution is to identify requirements, to carry out a design sufficient to get a blackbox description of the system, to identify and analyze hazards, and then to begin the lifecycle again with changed requirements. It may be necessary to continue to do so for each kind of system that is not yet so completely and thoroughly understood that it can be bought safe off the shelf or routinely put together out of pieces that can be bought safe off the shelf.

Then, it becomes necessary to accept that designs are an integral part of requirement specification in some domains or that at least that it may be essential to carry out enough of a design to see the full implications of the requirements so far, in order to see what the requirements should be. It is necessary to accept that such designs, like prototypes built for requirements exploration, understanding, and validation, are going to be thrown away or at least not used fully or without change.

References

[1] Anderson, T., de Lemos, R., and Saeed, A., "Analysis of Safety Requirements for Process Control Systems," in *Predictably Dependable Computing Systems*, ed. B. Randell, J.C. Laprie, B. Littlewood, H. Kopetz, Springer, Berlin (1995).

[2] de Lemos, R., Saeed, A., and Anderson, T., "On the Safety Analysis of Requirements Specifications," pp. 217–227 in *Proceedings of the Thirteenth International Conference on Safety, Reliability, and Security (SAFECOMP '94)*, ed. V. Maggioli, Springer (1994).

[3] Leveson, N.G., *Safeware: System Safety and Computers*, Addison Wesley, Reading, MA (1995).

[4] Leveson, N.G., "Intent Specifications: An Approach to Building Human-Centered Specifications," Technical Report, Computer Science and Engineering, University of Washington, Seattle, WA (1997).

[5] Modugno, F., Leveson, N.G., Reese, J.D., Partridge, K., and Sandys, S.D., "Integrated Safety Analysis of Requirements Specifications," *Requirements Engineering* **2**(2), p.65–78 (1997).

[6] Saeed, A., de Lemos, R., and Anderson, T., "Robust Requirements Specifications for Safety-Critical Systems," pp. 219–229 in *Proceedings of the Twelfth International Conference on Safety, Reliability, and Security (SAFECOMP '93)*, ed. J. Górski, Springer (1993).

[7] van Lamsweerde, A. and Letier, E., "Integrating Obstacles in Goal-Driven Requirements Engineering," Technical Report, Catholic University of Louvain, Department of Software Engineering, Louvain (1997).