Validation of Object Oriented Software Design With Fault Tree Analysis

Massood Towhidnejad
Department of Computing
Embry-Riddle University
Daytona Beach, FL 32114
towhid@erau.edu

Dolores R. Wallace
SRS Information Services
Software Assurance Technology Center
NASA - Goddard Space Flight Center
dwallac@pop300.gsfc.nasa.gov

Albert M. Gallo, Jr.
Code 304
Software Assurance Technology Center
NASA - Goddard Space Flight Center
Albert.M.Gallo@nasa.gov

Abstract

Software plays an increasing role in the safety critical systems. Increasing the quality and reliability of the software has become the major objective of software development industry. Researchers and industry practitioners, look for innovative techniques and methodologies that could be used to increase their confidence in the software reliability. Fault tree analysis (FTA) is one method under study at the Software Assurance Technology Center (SATC) of NASA's Goddard Space Flight Center to determine its relevance to increasing the quality and the reliability of software. This paper briefly reviews some of the previous research in the area of Software Fault Tree Analysis (SFTA). Next we discuss a roadmap for application of the SFTA to software, with special emphasis on object-oriented design. This is followed by a brief discussion of the paradigm for transforming a software design artifact (i.e., sequence diagram) to its corresponding software fault tree. Finally, we discuss challenges, advantages and disadvantages of SFTA.

1. Introduction

Fault Tree Analysis (FTA) [1] is a technique used in the area of reliability. Initially, FTA was introduced in the 1960s, with the primary purpose of identifying those circumstances that could cause a system to reach a hazardous or unsafe state. FTA is powerful static analysis tool. Given a specific hazardous state, FTA uses a *backward* (also referred as *top-down* or *deductive*) search technique in order to identify conditions that would cause the system to reach that state. In other words, once a specific hazard is identified (hypothesized), FTA will search all possible combinations of the conditions (initial states) that could force the system to reach that state. FTA is a graphical analysis tool and uses two techniques in its analysis: qualitative and quantitative. Through the qualitative technique, FTA is capable of identifying all possible combinations of conditions that would cause the system to reach a hazardous state. These combinations of conditions are referred to as *a cut set*. A *minimum cut set* represents a minimum number of conditions that need to be satisfied in order to force the system into a hazardous state. The quantitative approach uses probability information associated with each condition (initial state) in order to calculate the probability of occurrence of the specific hazardous state. One of the advantages of the FTA is the fact that all attention is paid to a specific hazardous state and identification of preconditions that need to be satisfied in order to reach such a state. Of course, this could also become a disadvantage if FTA is the only technique used to identify hazardous states. This is due to the fact that it is possible for the analyst to overlook a specific hazardous state. In order to prevent this situation, other techniques such as Failure Modes and Effects Analysis (FMEA) [2], a *forward* (also referred as *bottom-up* or *inductive*) search, need to be used in conjunction with the FTA to identify all possible hazardous states for a system.

FTA is typically applied to hardware systems, but recently attempts have been made to apply FTA to software. Section two elaborates on previous research in the area of the Software Fault Tree Analysis (SFTA), also a road map to application of Fault Tree (FT) throughout the development life cycle is presented. Section three presents application of the fault three to object oriented design. Section four briefly discusses the paradigm for conversion of the Unified Modeling Language (UML^{TM}) diagram to its corresponding FT. Finally, the last section of this paper discusses some of the advantages and disadvantages of application of the FT to software.

2. Software Fault Tree Analysis

There has been significant research on software fault tree analysis, with the majority having been conducted by Leveson [3], Lutz [4], and Dugan [5]. In most cases, however, SFTA is used at the code level, and the size of the software (measured by lines of code) to which the SFTA has been applied, is relatively small, approximately one thousand lines of code. Leveson [6] has generated a set of templates that could be used in SFTA, where a specific language construct (syntax) has been represented in the form of fault tree. It is important to mention that

Proceedings of the 28th Annual NASA Goddard Software Engineering Workshop (SEW'03)
0-7695-2064-2/03 $ 17.00 © 2003 IEEE

when FTA is applied to software, and specifically at the code level, we are only addressing the qualitative analysis, since at this level quantitative analysis does not make sense. Therefore, at implementation (coding phase), the objective of using SFTA is to identify the set of instructions that could possibly cause the software to reach a hazardous state. Therefore, one could use SFTA in combination with formal code inspection in order to increase their confidence in the safety of the software under investigation. Finally, it has been pointed out by a number of researchers that SFTA shows some weaknesses when there are loops involved in the code, but loops are almost always present in software. Therefore, this is a weakness that needs to be overcome. Additional work by some researchers like Helmer [7], and Modugna [8] resulted in the application of the SFTA to requirements with some success in the detection of the weak or missing requirements.

2.1 Application of SFTA during software development life cycle

Researchers and practitioners generally agree that applying SFTA at the code level is a very cumbersome and labor-intensive activity. In addition, it is a well-known fact that defect detection and correction at the implementation phase is much more costly than at the earlier stages of the software development life cycle.

Given this rationale, the SATC team recommends applying SFTA to requirements and design. The process is to use SFTA during the requirements and design phase to identify the critical component of the software where safety and hazardous states are the major concerns. Then SFTA may be applied at the code level only for these critical components. The above approach follows the principle of divide and conquer, which is one of the fundamental methods of solving problems. By partitioning the system (to the safety critical component and those that are not safety critical), we narrow the scope of the area in which FTA has to be applied. Of course it is assumed that special attention is given to the flagged components (i.e., safety critical partition) during the development and verification and validation activities.

SFTA at requirements phase

The main objectives of applying SFTA during this phase of software development are to:

- Identify weaknesses that exist in the requirement specification. Weak requirements will either be modified or additional requirements will be added in order to eliminate or mitigate these weaknesses.
- Identify all the requirements that have a direct effect on the safety of the system. This can be done either through the knowledge collected as part of the requirements elicitation, or identifying the pattern of use and the surrounding environment that could affect the software, by forcing it to a hazardous state. Once requirements with safety considerations are identified, these requirements will be traced throughout the development life cycle. It is assumed that a requirement traceability matrix is included in the software development artifacts to help with this task.

SFTA at design phase

The main objectives of applying SFTA during this phase are to:

- Identify the weaknesses of the high-level design. At this stage, appropriate modifications will be implemented in order to strengthen the overall design.
- Identify the components/modules and subcomponents that have direct effect on software safety. These modules and those implementing the requirements with the safety consequences are identified. Then, special attention may be given to the generation of their implementation, by guaranteeing the elimination of design factors that could force the system into a hazardous state.

The details of the application of SFTA during the design phase are discussed in Section 3 of this paper.

SFTA at implementation phase

The main objective of applying FTA to code is to identify critical code components that have direct bearing on the safety of the software. In this phase, fault trees will be generated for all the modules previously identified (during the detailed design phase) as critical modules affecting software safety. The goals here are to:

- Identifying a set of key instructions that could place the system in a hazardous state
- Add appropriate safeguards that prevent the software from reaching such a state.

As previously mentioned, the majority of the previous research in SFTA has been applicable to this phase of the software development.

One of the major advantages of the above approach is to avoid generating fault trees unnecessarily for significant amounts of code in the system. It limits the application of FT to small, but critical portions of the code that affect the safety of the software. Applying FTA to the entire system requirements specification and the detailed design phase will be much more efficient than broadly applying it at the code level. Another advantage of this approach is that by applying SFTA at every stage of development, safety issues are identified early in the development life cycle and remedies can be implemented as early as possible.

3. Application of SFTA to Unified Modeling Language Artifacts

Applying SFTA during the detailed design phase will produce the best return on investment. It is here that a software product exists in its most ideal form for SFTA to be applied. Software is represented in the form of some number of modules where functionality, interfaces, inputs, and outputs are well defined. This is the closest we get to representing software structure in a way that is analogous to hardware modeling, a point prior to development where the salient system features, i.e., gates, encoder, functionality, interfaces, inputs and outputs are well defined. The same can be said about a software system at the detailed design phase. Here the software is represented with an equivalent amount of detail that we can achieve the equivalent degree of insight. Applying SFTA at this point enables us to identify modules (objects, methods, or functions) that could directly affect the safety of the system.

In both the preliminary and detailed design phases, once a module or a set of modules is identified as having possible impacts on the safety of the system, additional safeguards need to be embedded into the design in order to guarantee their safe operation. It is worth mentioning again that generating fault trees for the system at this point will be a much more efficient choice than generating them during the implementation phase.

With the exception of Pai's work [9] on dynamic fault trees for systems, we were unable to find any previous dynamic work that applied SFTA during the design phase. The SATC team chose the Object Oriented Design (OOD) methodology as the vehicle for the application of SFTA at the design level. There are two primary reasons for choosing OOD: 1) much recent software design uses OOD and the designs are implemented using OO languages, and 2) recently many OODs use the UML^{TM} (Unified Modeling Language), which is standardized and commonly used by the software development community [10].

UML^{TM} uses a number of views and diagrams to describe software systems. The problem is how to relate these to the notation used in FTA. As the first step, we looked at all the different UML^{TM} diagrams and identified those we believe best match the SFTA. During this process, we identified the activity, sequence and state diagrams, as the first candidates for the application of SFTA.

Communicating and validating critical system details becomes challenging, to say the least. This is because most end users are not familiar with OO design artifacts such as graphs and diagrams; however, the majority of customers in the aerospace industry are familiar with hardware, they are generally comfortable with logic diagrams, which is the fundamental concept behind fault trees. Even in those rare instances where customers are unfamiliar with the concepts behind logic diagrams, it is relatively easy to achieve a comfort level with a handful of logic gates in a sequence diagram. These findings suggest that SFTA should be used not only as a verification technique for the software design, but also as a communication vehicle with customers.

Our work also indicates that customers, after reviewing a fault tree, easily detect the occurrences of missing design components. By pointing out these missing components, they are actually completing the fault tree, thereby improving quality of the design as well as the ultimate system.

Initially we applied SFTA to the activity diagram [11]. While we learned that it is possible to apply SFTA to the activity diagram, we also learned that special care is needed in order to handle any loop in an activity diagram. There is some ongoing research in this area [12], which appears promising; however, much work is still needed in this area.

We then attempted to apply SFTA to the sequence diagram, at which point we came across additional findings. We learned that while SFTA may serve as a technique for verification of design, it could also serve as a vehicle for improved communication with customers and other stakeholders. We have developed a partial paradigm for transforming sequence diagrams into software fault trees, which is discussed in more detail in section 4 of this paper.

Ultimately, we applied SFTA to the state diagram. We arrived at the same set of observations as in the case of sequence diagrams. Figure 1 represents the state diagram for a pay at the pump system, with its corresponding fault tree diagram represented in Figure 2. As noted for activity and sequence diagrams, special care must be given when representing timing constraints and occurrences of iteration.

4. Transformation Paradigm

One of the original objectives of this project was to investigate the feasibility of automatically generating FT for software design artifacts. However, after further investigation, it became obvious that complete automatic generation of the fault tree from a software design artifact, is neither feasible, nor desirable. However, it is possible to generate a minor (even less than the activity diagram) part of the fault tree automatically. The main reason is the fact that, if we are able to automatically transform software design artifacts to fault tree, then the defects, or the missing components, that are already in the design will also be transformed, which as a result defeat the purpose of having the fault tree analysis. In addition, as we increase the role of automation in this transformation process, we will decrease the analytical activity of engineers involved in the process of fault tree analysis.

Therefore, rather than looking at the automation process for transformation of the design artifact to its corresponding fault tree, we started looking at generation of some guidelines that an engineer may apply, when they are interested in generating a fault tree for a design artifact. The remainder of this section briefly describes some of the guidelines that could be used, when one is interested in generating a fault tree for a sequence diagram.

Guidelines for transformation of sequence diagram to fault tree

Our investigation revealed some limitations in translating a sequence diagram into a fault tree. We categorize these limitations to two categories, timing and iteration. The first category addresses the representation of the timing issues in the sequence diagram. Typically, there are two major timing situations. In the first, one event must be completed before a second can start. In the second case, an event must start but need not complete before a second is initiated. There are efforts in the research community [13] assessing the timing problem, where a set of augmented logic gates can be used to enforce the appropriate sequencing. The second category addresses the representation of the iteration and loops that are present in the sequence diagram into the corresponding fault tree. The following bullets describe how one could address these problems.

- Identify each object that affects the hazard, and represent each one as either a basic or an intermediate event. Each of these events feeds into an OR gate that generates the hazard under investigation (Fig. 3-I).
- Typically, any message (e.g., create, show.) that is affecting the hazardous situation is represented as an input to the OR gate, which its output feeds to the object that sent and received that message. It is possible for the object to fail to send that message or receive it; therefore, it should be represented as a basic event to the sender and receiver object (Fig. 3-II).
- When an object is created by another object, the created object is represented as a basic or an intermediate event (Fig. 3-III).
- The timing problem in the sequence diagram has generated a challenge for the development of the fault tree. Typically, there are two major timing situations. In one case, event A has to be completed before event B can start, and the second case is when event A has to start, but it does not necessary have to be finished before event B starts. Some research [13] has examined the timing problem, where a set of augmented logic gates can be used to enforce the appropriate sequencing; however, these augmented gates are not represented in anyone of the fault tree tools. For the time being, we can handle the timing problem in the following manner:
    - For the case where event A has to be completed before event B, we can AND the outcome of event A with the event B to enforce this timing sequence (Fig. 3-III).
    - To handle sequencing of two events (objects) concurrently active (e.g., it is required for the first object to become active before the second object becomes active). For example, object A instantiates object B, and then both objects perform concurrent activities. In order to represent this timing sequence, we can represent object A with two sub-objects (A1 and A2), where sub-object A1 represents the activity that is required to be completed up to the instantiation of object B, and sub-object A2 represents object A's activity starting with the instantiation of object B. As a result, the relationship between sub-object A1 and B is the same as what is described in previous bullet. By following this approach, we realize that we limit the timing constraint between the sub-object A1 and B, and as a result, there is no timing constraint between the sub-object A2 and B since they are active concurrently.
- If a message is sent to/from multiple objects, this can be represented via transfer logic gate, which is shown by a triangle (Fig. 3-IV).
- As previously mentioned, representation of the iteration by fault tree is a challenging task. Iteration (i.e., multiple calls to self) affects the fault tree from the quantitative point of view. There is no additional gate needed to represent the iteration; however, it affects the failure model. For example, if an object has an effect on a specific hazardous state, it does not matter if it is called once or many times; however, if it is called multiple times, then its quantitative effect increases. Therefore, using this approach, we would need to represent the reliability model of the component, when it is done once, and then incorporate another reliability model when it is done N times. For example, if call A has the failure probability of 0.1%, and it is possible for Call A to be repeated 50 times, then obviously, the probability of the failure for 50 times is no longer 0.1 %.

These conclusions confirm the strength of SFTA for design verification and effective communication with stakeholders.

5. Conclusions

It is important to point out that the main goal of our activities was the generation of the fault tree for a specific design artifact. As discussed, FTA comprises two distinct analyses – qualitative and quantitative. It is the former that reveals the minimum cut set which identifying the fewest number of conditions needed for the system to reach a specific hazard.

Our work shows that it is possible not only to generate the fault tree, but also to identify minimum cut sets; however, a major challenge still remains regarding quantitative analysis of the fault trees - specifically, the lack of dependable reliability data for software components. Researchers such as Musa [14] and Smidts [15] are currently working on this issue; however, much more work is needed in this area.

Diverse user communities could use SFTA for different purposes:

- Developers/Designers: Use SFTA as a tool to improve the product under development. They can also use it as a vehicle for improved communication between themselves and the customer.
- Quality Assurance: Use SFTA for the purpose of validating the product.
- Managers: Use the SFTA for risk analysis, decision support, and identification of areas needing special attention.
- Testers: Use SFTA for the purpose of planning their testing activities and focusing on areas needing additional stress testing. Testers also may use the fault tree for the purpose of the validation and verification.

There are a number of advantages and disadvantages related to the SFTA. These include:

- Advantages
    - Easy to learn and use
    - Graphical Representation
    - Communication vehicle with customer
- Disadvantages
    - Conversion is labor intensive
    - Lack of software reliability data
    - Timing and loops need special attention
    - No dedicated commercial SFTA tool available.

References

[1] NUREG-0492, Fault Tree Handbook, U.S. Nuclear Regulatory Commission, January, 1981.

[2] “Applying Integrated Safety Analysis Technique (Software FMEA and FTA)”, Jet Propulsion Laboratory, November 1998.

[3] Leveson, N. G. and P. R. Harvey, “Analyzing Software Safety”, *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 5, September 1983, pp. 569-579.

[4] Lutz, R. R., “Targeting Safety-Related Errors During Software Requirements Analysis,” *Journal of Systems and Software*, 1996, 34, 223-230.

[5] Dugan J.B., Sullivan, K. J., and D. Coppit, “Developing a High-Quality Software Tool for Fault Tree Analysis”, *Transactions on Reliability*, December 1999, pp. 49-59.

[6] Leveson, Nancy G., Stephen S. Cha, Timothy J. Shimeall, "Safety Verification of Ada Programs Using Software Fault Trees." *IEEE Software*. pp 48-59.

[7] Helmer, G., Slagell, M., Honavar, V., Miller, L. and Lutz, R., "A Software Fault Tree Approach to Requirement Analysis of an Intrusion Detection System" *Symposium on Requirements Engineering for Information Security,* March 5-6, 2001.

[8] Francesmary Modugno, Nancy G. Leveson, Jon D. Reese, Kurt Partridge and Sean D. Sandys, ``Integrated Safety Analysis of Requirements Specifications'', *IEEE International Symposium on Requirements Engineering*, 1997.

[9] Pai, G., J., and J. B. Dugan, “Automatic Synthesis of Dynamic Fault Trees from UML System Models”, *The 13th International Symposium on Software Reliability Engineering*, IEEE Computer Society, Annapolis, MD, USA, November, 2002, pp. 243- 254.

[10] OMG Unified Modeling Language, Version 1.4, September, 2001, http://www.omg.org/cgi-bin/doc?formal/01-09-67

[11] Towhidnejad, M., Wallace, D., Gallo, A. ““Fault Tree Analysis for Software Design”, 27th Annual IEEE/NASA Software Engineering Workshop, December 2002.

[12] Giarambino, “How to Avoid the Generation of Loops in the Construction of Fault Trees”, *Proceedings of Annual IEEE Reliability and Maintainability Symposium,* 2002.

[13] Helmer, G., et al., “Software Fault Tree and Colored Petri Net Based Specification, Design and Implementation of Agent-Based Intrusion Detection Systems”, http://www.cs.iastate.edu/~honavar/Papers/CPN-IDS.pdf, June 2002

[14] Musa, J., Software Reliability Engineering, McGraw-Hill, New York, 1999.

[15] Smidts, C., Sova, D. “An Architectural Model for Software Reliability Quantification: Sources of Data”, Reliability Engineering and System Safety, 64, 279-290, 1999.

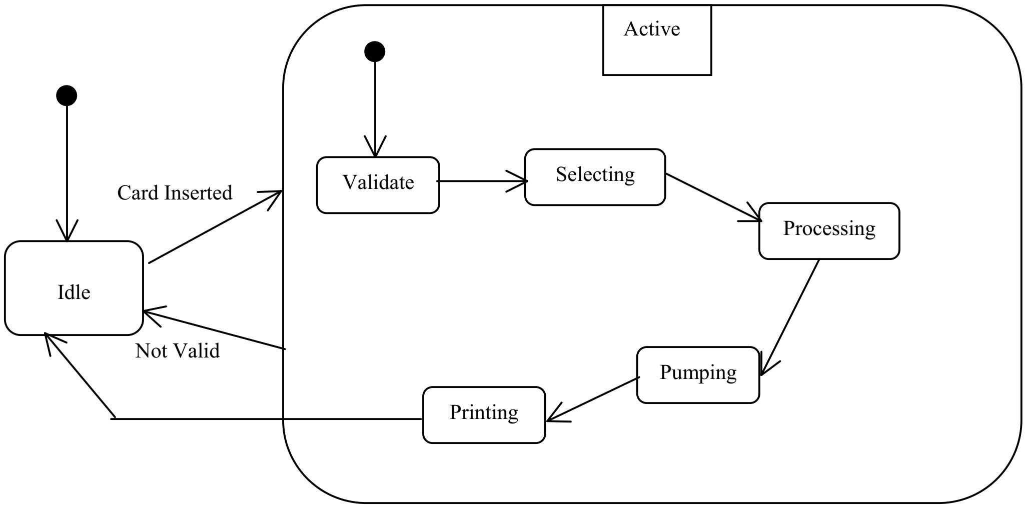


Figure 1. State Diagram for Pay at the Pump

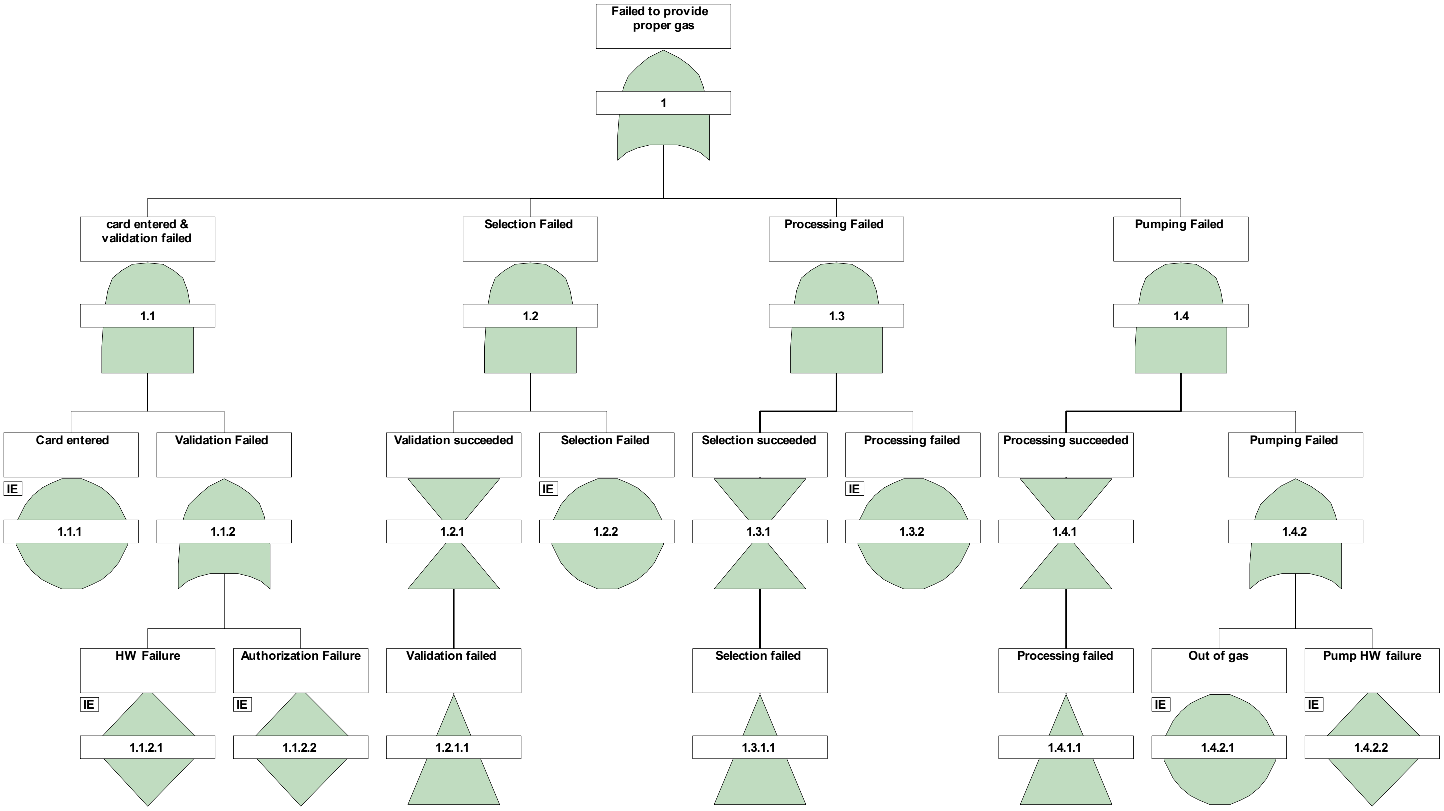
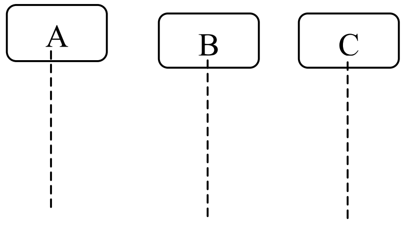
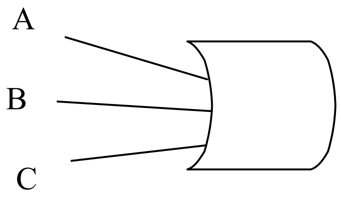
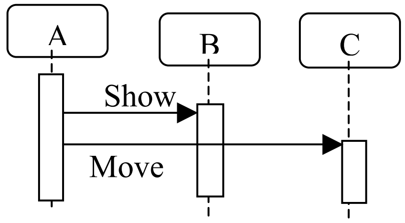
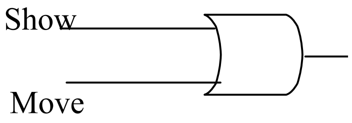
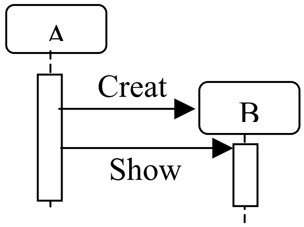
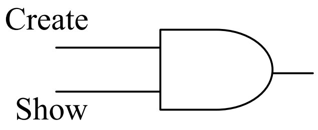
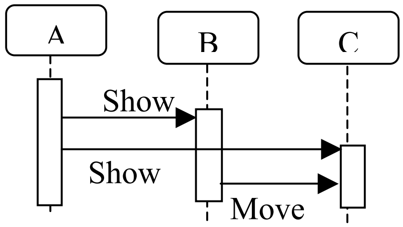
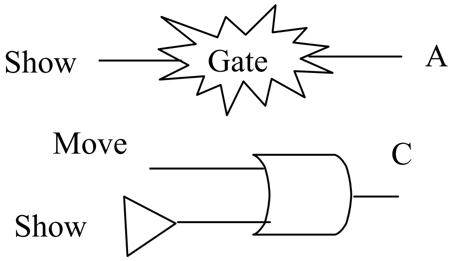


Figure 2. Fault Tree for Pay at the Pump

Fig.3	Sequence Diagram	Fault Tree
I		
II		
III		
IV		

Figures 3 I-IV. Sequence diagram transformation to software fault tree