Modeling Techniques for a Risk Analysis Methodology for Software Systems

Jim Wang
June 2003
CMU – ISRI – 03 – 101

Master of Science in Information Technology
in Software Engineering

Project Practicum

Institute for Software Research International
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213-3890

Faculty Advisor: Dr. James E. Tomayko,
Professor David Root

External Supervisor: Dr. Tsong-lun Chu
Brookhaven National Laboratory

Keywords: System Specification, System Modeling, UML, Z, EVES, Fault Trees

Abstract

The U.S. Nuclear Regulatory Commission (NRC) Office of Nuclear Regulatory Research is interested in developing approaches towards analyzing digital instrumentation and control (I&C) systems for nuclear power plant system upgrades. (Arndt 2002) These approaches are directed towards analyzing the changes in risk involved with using digital systems, which include software and hardware concerns. The purpose of this document is to outline possible techniques to use in a possible analysis methodology and to briefly describe how these techniques are used. This document suggests that a descriptive visual model be created using many of Unified Modeling Language's (UML) numerous artifacts. UML's widespread acceptance and comprehension, along with the possibility that UML documents have already been written for the system in question, make it an obvious choice for this application. In addition, a formal specification should be written in Z and analyzed with the help of the Z/EVES software package. Formal specification and analysis of that specification would ensure that the system is complete and that predetermined conditions hold throughout the operation of the software system. Finally, dynamic fault tree analysis should be performed on the system to analyze each hazard or failure event. Fault tree analysis is a technique that the NRC is very familiar with and adept at performing.

Table of Contents

Introduction

The U.S. Nuclear Regulatory Commission (NRC) Office of Nuclear Regulatory Research is interested in developing approaches towards analyzing digital instrumentation and control (I&C) systems for nuclear power plant system upgrades. These approaches are directed towards analyzing the change in risk involved with using digital systems, which include software and hardware concerns.

The purpose of this document is to compile a list of strategies and techniques for a possible risk analysis methodology that will include qualitative and quantitative modeling of the system, as per the NRC's requests, formal specification, and fault tree analysis. It will not go into detail as to the actual implementation techniques but will only briefly summarize the techniques suggested and provide some example scenarios.

Methodology Requirements Overview

In numerous meetings, papers, and discussions, the NRC has come up with several key requirements of a PRA methodology. At a high level, the requirements for the methodology are that it must:

a) Qualitatively model the digital I&C portions of identified accident scenarios with sufficient detail and completeness such that subsequent (non-I&C) portions of the scenario may be properly analyzed and practical decisions may be formulated and analyzed.
b) Quantitatively identify the likelihood of system failure in a credible manner.

It is worth noting that qualitative modeling should not be confused with qualitative analysis of a software system. Qualitative analysis is an analysis of a software system with respect to quality attributes and the tradeoffs between them. Quality attributes are attributes of a system such as reliability, performance, and maintainability; in system analysis quality attributes are used to assess design tradeoffs and help make decisions based on attribute priority. Qualitative modeling requirements in this case do not refer to those attributes.

Qualitative Modeling Requirements

Qualitative modeling techniques are commonly used when the information about the system component internals are only partially known. With software systems, it is not possible for every fault to be correctly and completely identified, and thus in this manner there is information about the system that is unknown. By employing qualitative modeling techniques in conjunction with credible qualitative techniques, an analyst is therefore able to make informed decisions and analysis on a particular system. In this regard, the requirements on a qualitative model identified by the NRC are that it contains sufficient detail and completeness in the information it provides such that analysis can be performed.

The methodology must be compatible with the prevailing risk analysis method employed with nuclear power plant (NPP) probability risk analysis's (PRA), event tree/fault tree analysis. The methodology need not employ event tree/fault tree analysis but it must be compatible with them.

1. Input data compatibility – It must not requirement highly time-dependent, continuous plant state information.
2. Output data compatibility – Discrete system states, which will subsequently be related directly to component performance and/or operator actions dependent on the I&C systems, must be provided for analysis.
3. Internal model constraints – varying levels of system degradation must be modeled, no other constraint is set.

Quantitative Modeling Requirements

While qualitative models may be used to analyze systems where all the information is not entirely known, quantitative models must also be used because these models will be used in conjunction with qualitative models. While they rely on partial, incomplete information, it is the duty of the analyst to develop credible models and prove their credibility with regard to the provided information. The methodology must be accurate, complete, credible, and be able to alleviate concerns about its own uncertainty.

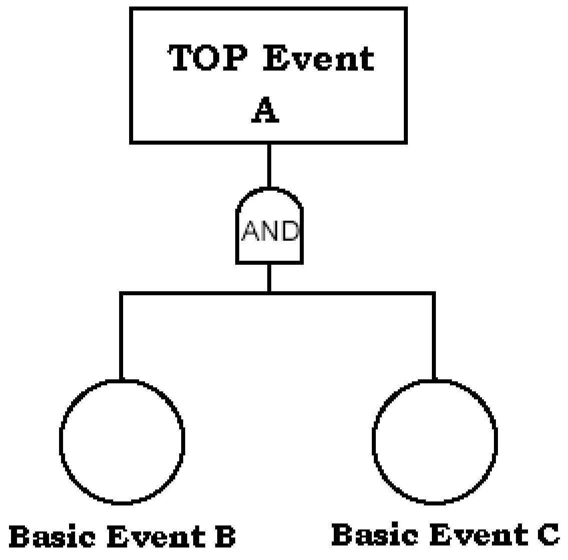
1. Accuracy – The NRC raises major concerns for covering common cause failure events and being able to explain them by;
    a. Directly quantifying the likelihood of basic events in a minimal cut set.
    b. and/or, adding basic events representing the different common cause failure events.
2. Completeness - The NRC asserts that proving a method's completeness is impossibility and only requires that a "good enough" method be developed.

3. Credibility – the methodology must be credible to the technical community and it must use alternative source of information such as data from private industry and applications and expert judgment.
4. Uncertainty – The methodology must mitigate uncertainties in its parameter values and with the model itself.

Fulfilling Qualitative and Quantitative Requirements

The qualitative requirements that state a model of the digital I&C portions of an accident scenario must be complete and should be satisfied by a sufficiently detailed fault tree analysis and UML artifacts. Proof of completeness is impossible because complete knowledge about all faults isn't possible, the team must model the faults that they do know about sufficiently in order to satisfy these modified completeness constraints. (Meshkat 2002) For example, if an analysis team wishes to investigate the consequences of a particular fault or hazard, they should examine every fault tree that contains the original fault as an internal node. The original fault themselves would be in their own fault trees, perhaps embedded in the current tree being analyzed, which would in turn be analyzed when investigating a failure. If the fault is related to a specific module, the team can then also examine UML artifacts such as the class diagram or sequence diagram to investigate the ramifications of the original fault.

As for the quantitative requirements, the same methods could be used to assign a quantitative value to the probability of a fault. In a fault tree analysis, the probability of a fault occurring is related to the failures in its leaf nodes.



For example, in the following above simple fault tree, if the probability of Basic Event B occurring is 0.5 and the probability of Basic Event c is 0.5, then the probability of TOP Event A occurring is 0.25. If the relationship were an OR (instead of an AND), the probability of the TOP Event A occurring would be 0.75. These values are calculated using mathematical probability rules.

Assigning probabilities to the two basic software events in the above example is subjective and depends greatly on the expertise of the analysis team. While there are methods for assigning probabilities to events, and the severity of failures, it still remains a difficult science. In some cases, historic, or even simulated, data on which to calculate failure probabilities is not available. Also, almost as important as the probability of failure, severity of failure is difficult to calculate for the same reasons.

Modeling

The first step in analyzing a system is to model that system. The system should be modeled in two ways. The first is a descriptive model that employs diagrams and figures to help visually explain how the system functions. These models will describe, qualitatively, the modules, and their interfaces, involved in the operation of the system and help clarify how the system operates. The second model should be a formal specification of the system in a language that can be analyzed with the use of a computer processor. Through these two methods, the system will be diagrammed visually for personnel use and specified electronically so analytical tests may be performed. UML is the suggested method for creating the descriptive model and Z, using EVES for analysis, is the suggested method for creating the formal specification, and subsequently analyzing that specification.

UML

The Unified Modeling Language (UML) is a methodology commonly used in private industry for modeling a software system. Developed, in part from prior work by others; and codified by James Rumbaugh, Grady Booch, and Ivar Jacobson (the “three amigos”), UML is a means of visualizing, specifying, constructing, and documenting artifacts for a software system. (Rumbaugh 1998) UML is the language of choice in many development processes, one of the most popular of which is Rational Unified Process (RUP) which is usually associated with the “three amigos,” for the very reasons it should be employed in modeling this system.

Usually, a collection of UML artifacts are created for a software system prior to its implementation in the software lifecycle and it is possible that these artifacts already exist for the system. These artifacts include, but are not limited to, class diagrams, use case diagrams, component diagrams, object diagrams, state-chart diagrams, and sequence diagrams. A brief description of each of the above named artifacts is located in Appendix A as well as an in-depth example of a use case diagram and a sequence diagram.

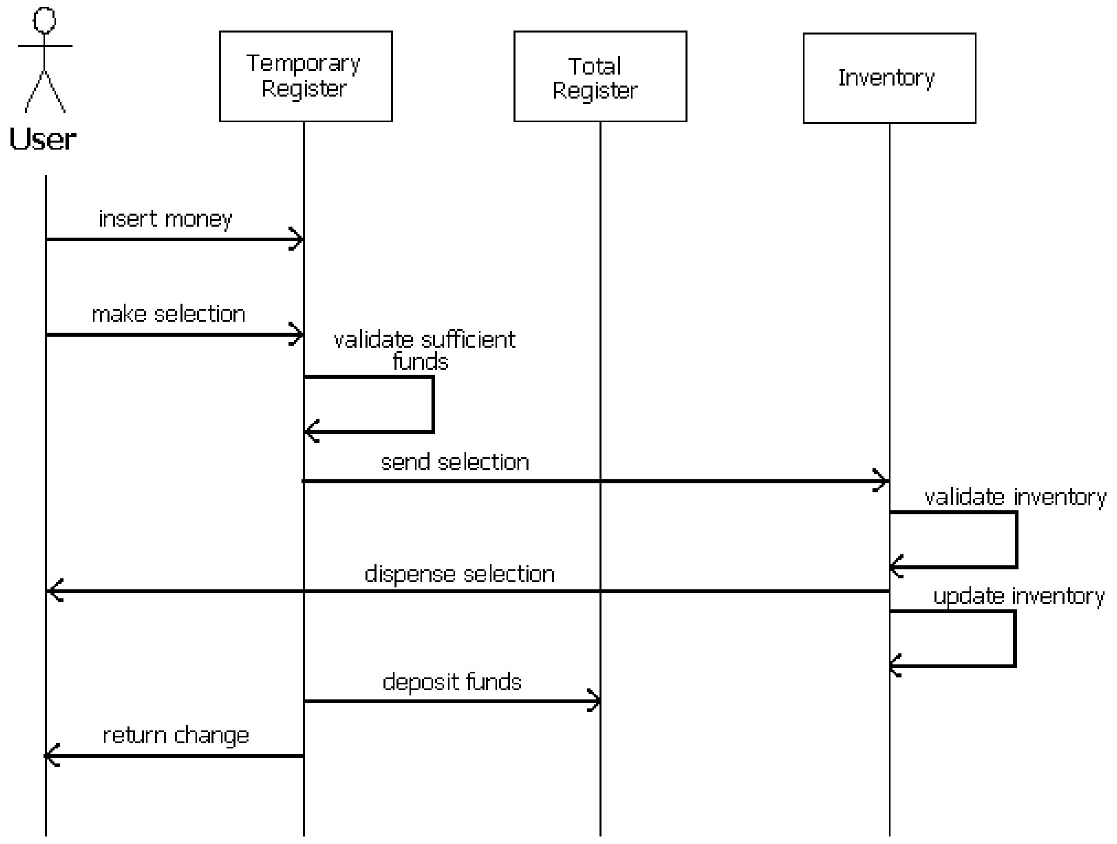


Figure 1: Instance Sequence Diagram

Figure 1 is an example sequence diagram written in UML. It illustrates the sequence of events when someone purchases a can of soda from a soda machine, but in general terms so that it could cover any vending machine. Figure 1 is called an instance sequence diagram, as opposed to a regular sequence diagram, because it describes only one possible sequence of events and is related to one use case. A particular system will have multiple use cases and multiple sequence diagrams (such as the sequence for a customer that doesn't insert enough money and subsequently wants their money back would be very different). In that situation a typical sequence diagram would describe a more general use case. Sequence diagrams are only one of the many UML artifacts at a developer's disposal in describing a system. In conjunction with other artifacts, these diagrams are very powerful.

The one of the advantages of using UML is that it is widely used and has a very shallow learning curve, as in the above examples about the vending machine. A developer unfamiliar with UML can understand UML artifacts with very little time because it uses very comprehensible language and diagrams in its basic form. In the example sequence diagrams, the user was depicted as a stick figure. UML is very useful in diagramming object-oriented systems, as it was designed for that purpose, and its many artifacts leave little of the system not explained. Another advantage is that because its wide usage, it is possible that the development firm that created the system also has UML documents with the necessary artifacts.

As with all modeling methods, there are disadvantages to using UML but many other methods suffer from the same disadvantages. UML artifacts are cumbersome to build and maintain for very complex systems. The only solution is the use of compartmentalization where components are contained within another in diagrams. Another is the existence of some redundant, useless, or arcane conventions and symbols that may lead to confusion.

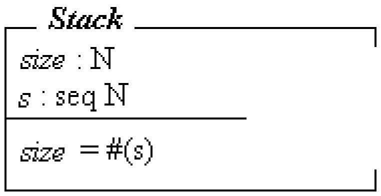
In general, however, many of the artifacts use simple language and require little explanation.

Actual documentation using UML can be as simple or as complex as necessary. There are numerous software packages for writing UML specifications and some are as complex as Rational Rose, by Rational Software, and others are as simple as WithClass, by MicroGOLD software. The complexity of your documents depends on the complexity of the system and the level specificity you wish to reach in documenting the system. A single class diagram could completely specify your entire system but it gives no information about events and the flow of the system. To illustrate flow, sequence diagrams and use cases scenarios would be necessary. More complex software packages such as Rational Rose can develop diagrams from other diagrams and interconnect objects between diagrams. An object represented by a class in a class diagram could be linked to a object in a sequence diagram and vice versa.

Z

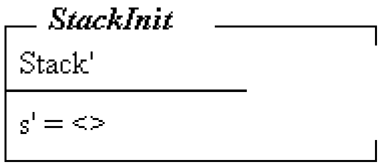
Z, pronounced “Zed,” is a formal specification language based on formal set theory and originally developed at Oxford by J. R. Abrial in 1979. Its basis in formal set theory is quite evident in the language and it does require a bit of training for someone to understand and use the language. While its rich history in academia has led to many implementations in that area, private industry has used it with success at companies such as IBM and Tektronics on safety critical projects. The purpose of Z is to produce schemas, the basic building blocks of the model, which represent entities or operations in the system; and build the entire system out of these. Z abstractly models parts of the system, such as pre- and post-conditions in a schema, and then performs analysis in the form of proving certain cases do or do not exist, such as deadlocks.

Schemas are quite simply to build and are represented by three parts. Each schema has a name, a data section, and an assertion section. The data section contains parameters for functions, subcomponents, and data such as variables or constants. The assertions section contains the invariants, pre- and post-conditions for the function or entity. Due to its set nature, schemas can be built from other schemas and usually are. This keeps schemas from becoming too cluttered but to the novice this may appear very confusing. The following is an example of a sample Z specification:



```
Stack
size : N
s : seq N
---------
size = #(s)
```

The above schema states that there is a Stack object in the space that contains two data items: its own size, as a natural number, and a sequence of natural numbers. Its size is defined as the number of elements in that sequence.



```
StackInit
Stack'
---------
s' = <>
```

The above schema represents the Stack initialization function, StackInit. It resets Stack (denoted by the prime notation) by setting s, the element holding the sequence of Naturals, to the empty sequence. StackInit contains all of the data and assertions of Stack but that component is represented by Stack'. This is the shorthand mentioned earlier that may be confusing.

$$\begin{array}{|l|}
\hline
\textbf{\textit{Push}} \\
\Delta \mathit{Stack} \\
e? : \mathbb{N} \\
\hline
s' = s \frown \langle e? \rangle \\
\hline
\end{array}
\qquad
\begin{array}{|l|}
\hline
\textbf{\textit{Pop}} \\
\Delta \mathit{Stack} \\
e! : \mathbb{N} \\
\hline
s = s' \frown \langle e! \rangle \\
\hline
\end{array}$$

The above schema represents the Push and Pop functions for the stack. They differ in that Push takes an element as input (denoted by the ?) and Pop returns an element as output (denoted by the !). The delta ahead of the Stack indicates that the Stack object is being changed by this function. In push an element is added to the stack, s, and in pop an element is removed.

$$\begin{array}{|l|}
\hline
\textbf{\textit{NumElements}} \\
\Xi \mathit{Stack} \\
\mathit{howMany!} : \mathbb{N} \\
\hline
\mathit{howMany!} = \mathit{size} \\
\hline
\end{array}$$

This final schema is a function that returns the size of the stack. Remember, the Χ *Stack* indicates that the original Stack schema is present but only read-only.

One critically important extension to Z was the additional of Schema Calculus by Spivey. This allowed the construction of larger schemas from smaller schemas. For example, in the above Pop function, there is no error detection or correction in the event that the stack was empty. A solution, using schema calculus, would be to create schemas for situations where the stack was empty and construct a new schema, *RobustPop*, for such situations.

$$\begin{array}{|l}\hline \textbf{\textit{IsEmpty}} \\ \Xi\, Stack \\ \hline size = 0 \\ \hline\end{array} \qquad \begin{array}{|l}\hline \textbf{\textit{IsNotEmpty}} \\ \Xi\, Stack \\ \hline size > 0 \\ \hline\end{array}$$

$RobustPop \mathrel{\widehat{=}} (Pop \wedge IsNotEmpty) \vee IsEmpty$

This would mean that RobustPop would either Pop an element or not pop an element, because the stack is empty; it would however, not, crash because of an empty stack. The IsNotEmpty schema is not necessary and was added for readability.

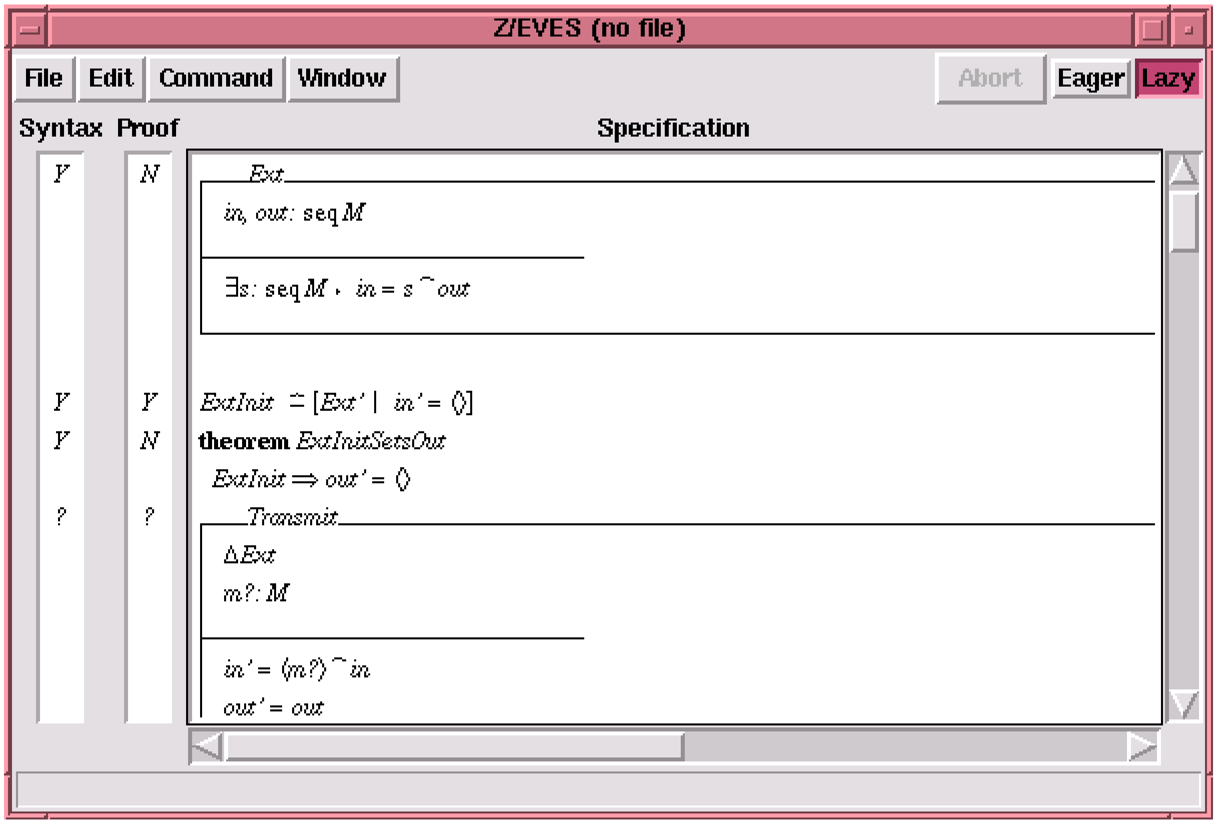
Z is more suited for systems that are state-orientated, as opposed to event-orientated, and allows for automated analysis with the use of software. There are other advantages to using Z that include its precision. Using software, one could perform automated analysis of the system with respect to it holding certain conditions. Using first order predicate logic, an assertion can be made about the system and using software, it can be shown whether that assertion holds true for the entire run of the system. While the complexity of the system and the complexity of the assertion affect how long the analysis runs, it can be done using software.

The major disadvantage of using Z is learning the language itself. It is based on set theory and thus if the user is not familiar with set theory then he or she must learn it in order to be successful with Z. This disadvantage is overcome once basic set theory is learned since the syntax of the language is straightforward. Another disadvantage of using Z is the same problem that hinders UML and practically any method of specifying or diagramming a system, complexity. In addition to the difficulty of building and maintaining the schemas for complex systems, as the complexity increases the running time of the condition checks when using software increases as well. Since software packages build trees representing states and events, as the schemas become more complex, the trees grow exponentially. While there is pruning and rebuilding of the tree to improve efficiency, the running time of these packages is still quite slow and cumbersome at times.

Analysis

Z/EVES

Z/EVES, as it is commonly called, is a software package that can be used to perform general theorem proving, type checking, and various other interesting functions on a Z specification. It is a powerful tool that can be used to do the sort of automated analysis mentioned earlier.



Syntax	Proof	Specification
Y	N	*Ext* $in, out : \mathrm{seq}\, M$ $\exists s : \mathrm{seq}\, M \bullet in = s \frown out$
Y	Y	$ExtInit \mathrel{\widehat=} [Ext' \mid in' = \langle\rangle]$
Y	N	**theorem** *ExtInitSetsOut* $ExtInit \Rightarrow out' = \langle\rangle$
?	?	*Transmit* ΔExt $m? : M$ $in' = \langle m? \rangle \frown in$ $out' = out$

Figure 2: Screenshot of Z/EVES

Figure 2 is a sample screenshot of Z/EVES with a system specified in Z. As you can see it automatically does syntax checking as you create the specification that is very helpful for Z novices.

Analysis in Z using Z/EVES takes the form of developing conditions and then using the software to verify that a condition holds on a system. A condition is stated using first order predicate logic using constructs such as the universal and existential quantifiers. For example,

□ x: S • P(x) – states that for all states of x in the system S, P(x) holds true.

□ x: S • Q(x) – states that for all states of x in the system S, there exists a state in which Q(x) is true.

A typical analysis would be to state a system invariant using the universal quantifier (□) and then using the software to verify that it holds true for all states. There are many more quantifiers and many intriguing ways in which they may be connected to state different conditions.

The advantages of using Z/EVES are that it provides automated analysis of the specification along with automatic syntax checking and other convenient features. A steep learning curve is one of the major reasons why the Z/EVES system, and many other academically developed formal methods, has not been used in private industry. Z/EVES itself requires time to learn and when coupled with the learning curve of the Z specification language, it is understandable why one would be hesitant to invest the time required to successfully use those tools.

Fault Tree Analysis

Developed in the early 1960's, fault tree analysis has become a widely adopted method of performing reliability and risk analysis in a wide range of industries. They provide a compact graphical visualization of the interaction of failures and events within a system and contain a logical flow to these events. Its hierarchical structure, with basic events at the bottom and identified hazards or failure modes at the top, lends itself to quick analysis with the use of logic symbol connectors. Through analysis, the probability of a particular failure occurring may be calculated from the probabilities of its events that cause it. In addition to being able to visually identify relationships and dependencies, many software packages are able to analyze fault trees.

The fault tree model presents a logical method for analyzing failure characteristics of a system and, equally importantly, shows that failure scenarios have been identified and which ones have not. After an analysis, high-risk areas, such as failure scenarios with high occurrence probabilities or high severity ratings, are identified and preventative measures may be employed to decrease the occurrence probability or the severity rating.

Dynamic Fault Trees

Fault tree analysis begins with the identification of all possible hazards or failure modes. Each possible hazards and failure mode will be further developed into its own individual fault tree through the use of Boolean logic gates and basic events. Dynamic fault trees differ from static fault trees in that they incorporate the notion of failure sequence. (Meshkat 2002) Traditional static fault trees cannot model failures in which the order of component failure is important as well as dynamic fault trees. An example of a sequential dependence is provided below:

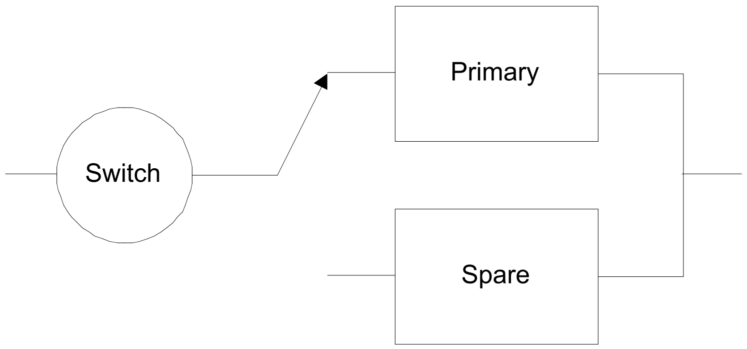


Figure 3: Sequential Dependence Example

In the above system, there are two components in a system; the spare is the backup to the primary component. If the primary component fails, the switch will "switch" to the spare component as a backup and operation of the system will continue. In the first failure situation, the primary fails before the switch fails. In this case, system operation continues since the switch may bring the spare component online. In the second case, the switch fails before the primary fails. The system is no longer operational, with respect to this component, because with the switch failing first, there is no way to bring the spare component online. In this case, the sequence of failure is important.

In the above situation, component failure would be modeled as:

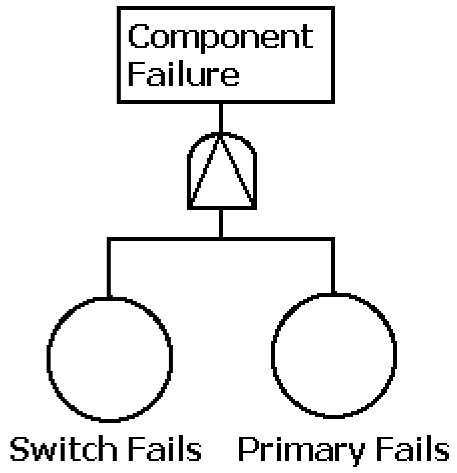


Figure 4: Priority-AND (PAND) gate

The priority-AND gate, modeled has a triangle within the regular AND gate, states that the above state only occurs if the first event happens before the second. The switch must fail before the primary fails before component failure occurs. While this situation can be modeled using static fault trees, it can be done more gracefully in dynamic fault trees using the PAND gates.

There are several difficulties with using fault tree analysis. It is assumed that every hazard and every failure mode will not be completely identified but proof of the best effort must be provided. In other words, completeness is not guaranteed and it is not possible to guarantee it. Possible methods of providing such proof include identifying that the risk contribution of any non-identified failure is small in magnitude both in probability of its occurrence and its severity. If a failure's possibility of occurring is very small or its impact on the system should it fail is very small, then the failure's significance is adjusted accordingly.

In addition to the question of uncertainty is the issue of complexity. As the system becomes more and more complex, the solution time of the analysis will increases exponentially. These are the same problems that any sort of automated system analysis technique must tackle and academia has been focused on increasing the efficiency of these types of software packages.

One such software package is Galileo developed by the University of Virginia. It uses DIF trees, or dynamic innovative fault trees, which uses binary decision diagrams (BDDs). BDDs are regarded as the best static fault tree solution technique (Anderson 1999) and it is what Z/EVES uses in its analysis. To improve efficiency with using DIF trees, sub-trees are divided up and solved prior to an analysis when possible. Galileo is a research prototype designed for use in personal computers and is the product of two professors at the University of Virginia, Prof. Joanne Bechta Dugan and Professor Kevin

Sullivan. One of the benefits of using Galileo is that it interfaces with Microsoft Word and Microsoft Visio quite gracefully.

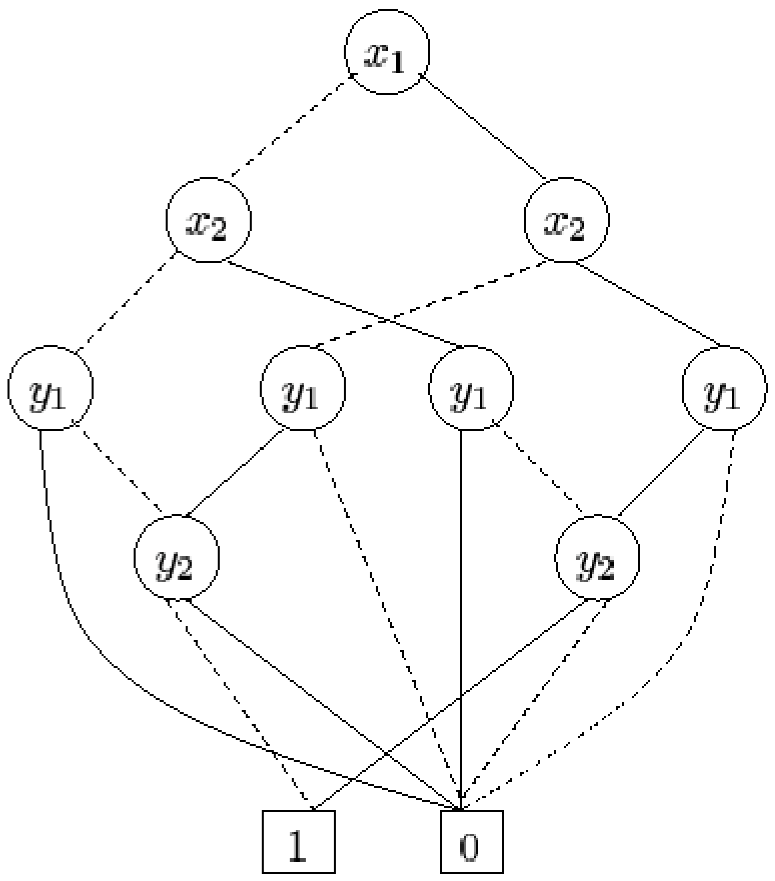


Figure 5: Reduced Binary Decision Diagram for (x_1 □ y_1) □ (x_2 □ y_2)

Figure 5 is a decision diagram for a logical expression, (x_1 □ y_1) □ (x_2 □ y_2), and through the use of software, it can be analyzed for its resultant value based on the values of the components. The efficiency of analysis depends on the manner in which the tree is constructed. The ordering in this case was x_1, x_2, y_1, y_3 and is written as $x_1 < x_2 < y_1 < y_2$. The tree itself contains nine nodes and cannot be simplified any further.

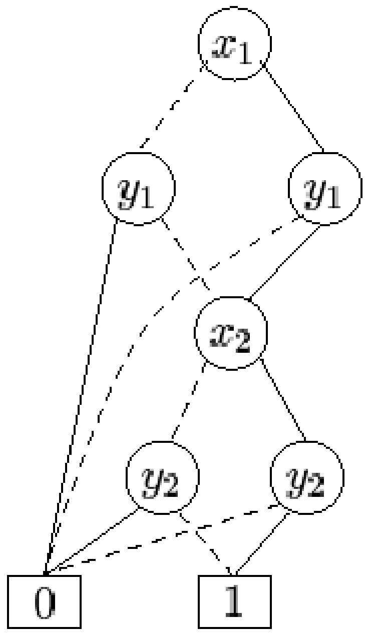


Figure 6: Another Binary Decision Diagram for $(x_1 \square\ y_1) \square (x_2 \square\ y_2)$

This ordering of variables, $x_1 < y_1 < x_1 < y_2$, produces a tree with only six nodes. It is clear that representing the same expression, $(x_1 \square\ y_1) \square (x_2 \square\ y_2)$, using three fewer nodes, approximately a third fewer, would greatly improve the efficiency of any algorithm.

Galileo's developers indicate that Galileo is a research prototype and thus is not as robust as would be necessary for safety critical systems such as those in the nuclear or medical field. This only indicates that Galileo should not be used alone when analyzing systems but when used in conjunction with other analysis methods such as those indicated earlier, Galileo is still very valuable as a tool.

Summary

There are various techniques for modeling and analyzing software systems and the requirements of the analysis team should be used in deciding which technique to use for each application. The system itself should also be used in deciding which techniques to use as well because in certain cases a particular type of system is modeled better using one technique instead of another. For example, Z is state based so a system with a significant number of events or a system centered on events would be more adequately modeled by another technique such as Petri Nets. The techniques outlined in this document serve only as a suggestion based on the requirements of the NRC.

REFERENCES

Ambler, Scott W. "The Elements of UML Style." Cambridge University Press. November 2002.

Arndt, S. A., E. A. Thornsbury, and N. O. Siu. "What PRA Needs from a Digital Systems Analysis." U.S. Nuclear Regulatory Commission, Washington. 2002.

Anderson, Henrik Reif. "An Introduction to Binary Decision Diagrams." The IT University of Copenhagen. 1999.

Clemens, P. L. "Fault Tree Analysis." JE Jacobs Sverdrup. February 2002.

Kalajdziski, Slobodan. "UML Tutorial in 7 Days." University "St. Cyril and Methodius" Skopje.

Meshkat, Leila. "Dependability Analysis of Grids Using Dynamic Fault Trees." Reliability Workshop. USC Information Sciences Institute. June 2002.

Rumbaugh, James, Ivar Jacobson, Grady Booch. The Unified Modeling Language Reference Manual. Addison-Wesley Publishing Company. New York. December 23, 1998.

Saaltink, Mark. “The Z/EVES System.” ORA Canada. Canada.1997.

Sullivan, Kevin. David Coppit, Ragavan Manian, and Joanne Dugan. “Combining Various Solution Techniques for Dynamic Fault Tree Analysis of Computer Systems.” University of Virginia Department of Computer Science. Virginia. 1999.

Woodcock, Jim and Jim Davies. “Using Z: Specification, Refinement, and Proof.” Prentice Hall. 1995.

Appendix A: Sample UML Diagrams

This appendix will elaborate on two of the many types of UML artifacts commonly used in describing software systems. There are many UML artifacts in use today and some books, such as Ambler's Elements of UML Style, go into depth about the guideliens for more than eight such artifacts.

The sample diagrams will follow the theme of modeling a very simple soda machine where a soda costs seventy five cents and the user may make one of two selections (Pepsi or Coke). This simple example will be enough to illustrate the benefits of UML.

Use Case Diagrams

Use case diagrams are used to illustrate how the system can be used. It is important to identify both internal and external aspects of the system such as which actors are involved in which processes. With the help of other diagrams, the different system flows can be illustrated easily using graphical means and prose.

Use Case 1: User inserts money and makes a valid selection.

This is probably the most common use case of the system, where the user inserts a sufficient amount of money for a soda and selects one in which the inventory is able to fulfill.

1. User inserts money into the machine.
2. The money is transferred to the temporary register.
3. The user makes a selection.
4. The machine validates that the temporary register contains sufficient money to make a purchase.
5. The machine indicates the user has inserted the sufficient amount.
6. The machine checks if the inventory is able to fulfill the selection.
7. The machine indicates the inventory is able to fulfill the selection.
8. Seventy-five cents is withdrawn from the temporary register.
9. The machine dispenses the selection.

10. Excess money is returned to the user via the change slot.

Error case 1: The user has not inserted enough money to make a purchase.

If after step 4, the user hasn't inserted enough money to make a purchase, a message is displayed indicated that there isn't sufficient funds to make a purchase. The user is then directed to insert more money or press the change return button to retrieve their money.

Error case 2: The user has made a selection that is out of stock.

If after step 6, the machine may not have the selection in inventory. It indicates to the user that the machine is out of stock and the user is directed to make another selection or press the change return button to retrieve their money.

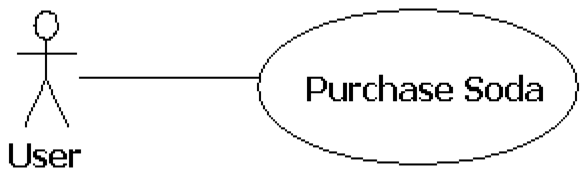


Figure 1: Simple Use Case Diagram

The above figure is quite a simple representation of the system but it is a valid UML diagram. It indicates that the User is outside the system and that no other actors are involved in the purchase of a soda.

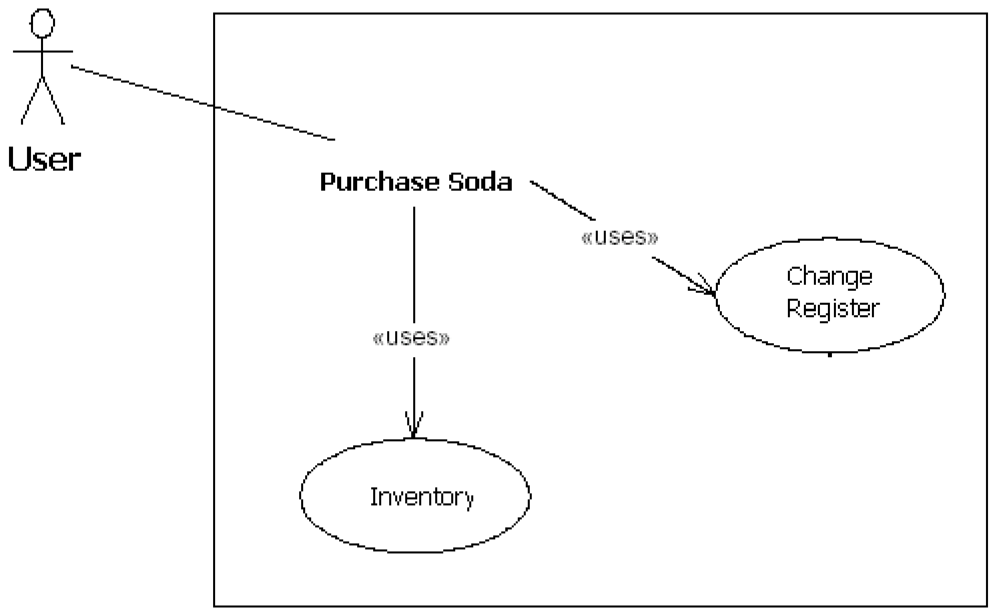


Figure 2: Use Case Diagram

The above figure is a little more descriptive than the first Use Case diagram and it indicates which objects within the system are used. You can see that in purchasing a soda, the change register and the inventory objects are used. In referencing the prose, you can see that the change register is used to validate enough money ahs been inserted and the inventory is used to validate that the selection is in stock.

<u>Sequence Diagrams</u>

Sequence diagrams are typically used to illustrate flow within a system. A sequence diagram will identify, in a specific sequence, which objects communicate with which other, and how, and in what order these communications must occur. The use case diagram itemized a list of events that occur in a particular scenario, the successful purchase scenario. It did not identify certain key factors such as how the objects communicated, this information is generally supplied by a sequence diagram.

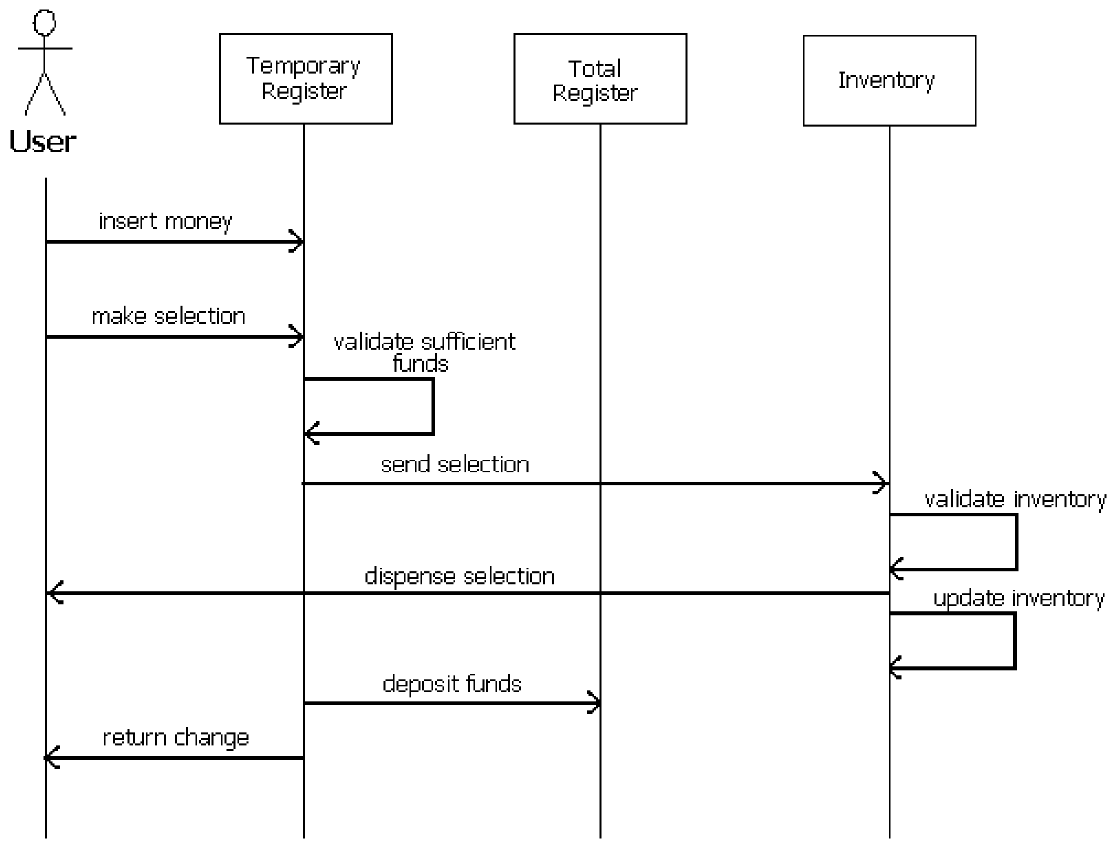


Figure 2: Sequence Diagram

This sequence diagram is similar to the one included in the body of the original document except it is not as specific as the example in the original document. The flow of the operations is still evident except exact function calls, and the attributes they require, are not included. The reason for this is because this is designed to be more general and more of an abstract look at the system. The objects (Temporary Register, Total Register, Inventory) do not correspond to classes, whereas in the more specific example they did, and the lines do not correspond to functions. As the first iteration of a sequence diagram, it is quite descriptive. Further iterations of the sequence diagram, along with the other diagrams, will elaborate the system.

There are many other artifacts used by UML practitioners such as class diagrams, component diagrams, and deployment diagrams. While there are many others, these are probably few of the most important diagrams and cover the greatest range of use within the software lifecycle for describing the system. Each diagram has a specific purpose and each one has its descriptive strengths and weaknesses. Below, each will be described in further detail and their strengths and weaknesses identified.

Use Case Diagrams:

Use case diagrams are the first diagrams discussed in this appendix but its strengths and weaknesses were not addressed. Use Case diagrams are very valuable in describing, in general terms, how the system operates. The prose component lists order of operations, from a high level, as well as possible fault conditions, when applicable, and system responses to those faults. The graphical diagrams are useful in building an overall view of system operation from an actor standpoint. Different conceptual components, such as a distributed cluster of computer processors, are treated as a single actor in the system, making the diagrams easy to understand and not cluttered.

The weakness is in the fact that it is a high level description of the system. It does not specify any components but it is a precursor and almost necessary artifact prior to the proper creation of other artifacts such as class diagrams.

Class Diagrams

Class diagrams are diagrams in which individual object classes are drawn and connecting using relational connectors. They resemble are a type of entity-relationship diagram with each class represented as an entity. Class diagrams are useful in that they give specific implementation descriptions of each class and their relationships to each other. A single class will describe the classes' attributes and interfaces, public and private.

The strength is in its specificity of the various objects in the system. A developer can take a class diagram and know almost exactly what he is building. More importantly, a developer can take the class diagram of a component and reuse that component based on the interfaces the diagram describes.

The weakness is that the class diagram assumes an object-oriented approach, the objects being the classes. If the system is not object oriented, if all functionality resided in one class, then a class diagram would not be as valuable. Also, the relationships themselves are not very descriptive other than indicating basic relationships such as "uses" or “contains.” How does class A “use” class B in a particular diagram? For that, further artifacts (such as a sequence diagram) would be necessary. Also, how is each function within the class implemented and how is each attribute used? For that, a system specification is needed.

Component Diagrams

In general, classes within a system can be grouped together to form discrete components. With any system, components may consist of one class or many classes. In many cases, the components are reusable if only their interfaces were known and easily managed. The component diagram artifact exists for this very purpose. Whereas a class diagram indicates relationships between classes, a component diagram indicates relationships between components.

Component diagrams are useful because they allow developers to use a component from one system with that of another, as long as they have a component diagram that describes the components various interfaces within the original system. Component diagrams are also useful because they act as the bridge between understanding a use case diagram and a class diagram. A class diagram is a very low level artifact and a use case diagram is a very high level artifact, a component diagram is between the two.

Its weakness is that it cannot standalone as a description of the system. It is not general enough and it is not specific enough to work with, it helps bridge the gap between class and use case diagrams when thinking of the system conceptually.

Deployment Diagrams

Deployment diagrams describe how a system is to be installed and deployed in a particular environment. For example, a deployment diagram for an http server (a software application that processes requests and serves text, images, and other assorted data types) would describe how the software would be installed on a single web server or a cluster of distributed web servers. Deployment diagrams are important because they indicate how the system is to be installed, used, and possibly maintained. Deployment diagrams also help in design decisions because, for the http server for example, a piece of software designed for one server is built differently than one designed for multiple intercommunicating servers. Outside of these purposes, deployment diagrams are not as helpful.

Appendix B: Z Specification

Introduction

Z is pronounced “Zed” and in common use it is written as either Z or Zed. The following is a Zed specification for the vending machine example. Zed is powerful as a specification, especially if you are familiar with logic and set theory, but it requires that prose explain the schemas in order to be complete and easily understood. No Zed specification is complete without the explanatory prose, which is found in bold below the schema.

Z/EVES uses the LaTeX language when specifying the schemas. The reason for using is because of LaTeX’s widespread use in academia and because the language lends itself to the way Zed is written (it contains the useful symbols and constructs used in Zed). In this example we will specify each schema in its original LaTeX form and in a graphical form that many may be used to seeing in print. The graphical form is not the graphical LaTeX output but is very similar to it. You will find that the graphical form is much more readable than the LaTeX form, especially if you are unfamiliar with LaTeX.

```
[ITEM, MSG, MONEY]
```

This system has three collections, ITEM, MSG, and MONEY.

```
ITEM ::= Nothing | Coke | Pepsi
MSG ::= "Dispensing…" | "Insert more money please" | "Out of Stock" |
        "Money accepted" | "Money not accepted"
MONEY ::= 5 | 10 | 25 | 50 | 100
```

The set of ITEM consists of Coke and Pepsi, corresponding to the possible dispensable items of the system. The Nothing item represents the case where nothing is dispensed from the machine. The set of MSG consists of the there messages that could be displayed by the machine. “Dispensing…” indicates the machine is dispensing an item, “Insert more money please” indicates that there is insufficient funds in the temporary register, “Out of Stock” indicates that the item the user selected is out of stock, “Money accepted” indicates that the inserted money is acceptable, and “Money not accepted” indicates that the inserted money is not acceptable. The set of MONEY represents the different denominations of money that may be entered into the machine, corresponding to the cent value of each. A nickel is 5, a dime is 10, a quarter is 25, a fifty-cent piece of 50, and a dollar is 100.

```
\begin{schema} VendingMachine
register_total : int
temp_register_total : int
num_coke : int
num_pepsi : int
\where
\end{schema}
```

The vending machine has two registers, one for the amount it's collected in its main register and one in its temporary register. The temporary register is where money is stored prior to a patron purchasing a soda. An int is used to record the number of cents in either register. Num_coke and num_pepsi store the number of Coca-Cola and Pepsi cans remaining in the vending machine.

There are several assumptions made for this system. The first is that the system transparently takes care of different coin denominations. It is also assumed that the registers are somehow connected so that when change needs to be dispensed, it can draw the necessary coins in the necessary denominations from either register. For example, if a dollar is inserted, a quarter must be returned but technically the temporary register doesn't have a quarter, it has a single unit which is the dollar. The quarter is thus drawn from the total register.

Another assumption is that currency not recognized by the machine, that is it is not in the set of MONEY, can be represented by an int and is treated as one with respect to when it is returned to the user.

```
\begin{schema} InitVendingMachine
VendingMachine
\where
register_total = 1000
temp_register_total = 0
num_coke = 20
num_pepsi = 20
\end{schema}
```

VendingMachine initialization is what occurs every time the machine is reloaded and cashed out. Ten dollars are left in the machine, in a set combination of three coin denominations (nickel, dime, quarter), in order to produce change.

```
\begin{schema} DispenseItem
\Delta VendingMachine
selection? : ITEM
change! : int
dispensing_item! : ITEM
message! : MSG
\where
(temp_register_total \geq 75) \land
      (((selection? = Coke \land num_coke > 0)
      \land (num_coke' = num_coke – 1))
      \lor ((selection? = Pepsi \land num_pepsi > 0)
      \land (num_pepsi' = num_pepsi - 1)))
      \land (dispensing_item! = selection?)
      \land (change! = temp_register_total – 75)
      \land (temp_register_total' = 0)
      \land (register_total' = register_total + 75)
      \land (message! = "Dispensing…")
\lor
(temp_register_total \geq 75) \land
      ((selection? = Coke \land num_coke = 0)
      \lor (selection? = Pepsi  num_pepsi = 0))
      \land (temp_register_total' = temp_register_total)
      \land (register_total' = register_total)
      \land (change! = 0) \land (dispensing_item! = Nothing)
      \land (message! = "Out of Stock")

\lor
(temp_register_total \leq 75) \land
      (temp_register_total' = temp_register_total)
      \land (register_total' = register_total)
      \land (change! = 0) \land (dispensing_item! = Nothing)
      \land (message! = "Insert more money please")
\end{schema}
```

DispenseItem is a function called when the patron selects what soda they wish to purchase. If the temporary register contains enough money, seventy-five cents, the function performs a series of steps to dispense the soda. First it checks to see what the user selected, then it checks the inventory to ensure the item is still in stock. If it is in stock, then the system dispenses the selection, adjusts the temporary and total registers, dispenses any necessary change, and displays a helpful message to the user. If the item is not in stock, then it displays a helpful message ("Out of Stock") and maintains the original value for each of the attributes. Finally, if the amount the patron has put into the machine is not sufficient (under seventy five cents), all attributes are maintained and a message is displayed (“Insert more money please”).

```
\begin{schema} InsertChange
\Delta VendingMachine
amount_inserted? : int
msg! : MSG
return! : int
\where
((amount_inserted? \in MONEY
      \land temp_ register_total’ = temp_register_total +
                                    amount_inserted?
            \land msg! = “Money accepted”)
\lor
(amount_inserted? \notin MONEY
      \land register_total’ = temp_register_total
      \land msg! = “Money not accepted”
      \land return! = amount_inserted?))
register_total’ = register_total
num_coke’ = num_coke
num_pepsi’ = num_pepsi
\end{schema}
```

InsertChange is a function called when the patron inserts money into the machine. If the money inputted is valid currency, the amount is added to the temporary register. If it is not valid currency then it is not accepted. All other attributes are unaffected.

```
\begin{schema} ReturnChange
\Delta VendingMachine
change! : int
\where
register_total’ = register_total
temp_ register_total’ = 0
num_coke’ = num_coke
num_pepsi’ = num_pepsi
change! = temp_register_total
\end{schema}
```

ReturnChange is a function called when the patron presses the return change button on the machine. Change is dispensed from the temporary register, thus setting it to zero. All other attributes are unaffected.

A Zed specification allows you to check whether certain contraints are ever violated. It can only tell you where the system can fail by design, not by some external causes. An example of an external cause would be if the power failed, the Zed specification does not cover that because it is not a design issue.

While this is a simplistic example, we are still able to create some constraints and check them against the system. Using first order predicate logic, we can specify constraints and check them against the model.

The first constraint we will state is that at no point in the operation of the vending machine shall it dispense an item when the temporary register holds fewer than seventy-five cents. This is stated as in FOPL (and LaTeX):

```
AG ((temp_register_total < 75) \land
      ((dispensing_item = Coke) \lor (dispensing_item = Pepsi))
```

and as this in plain English:

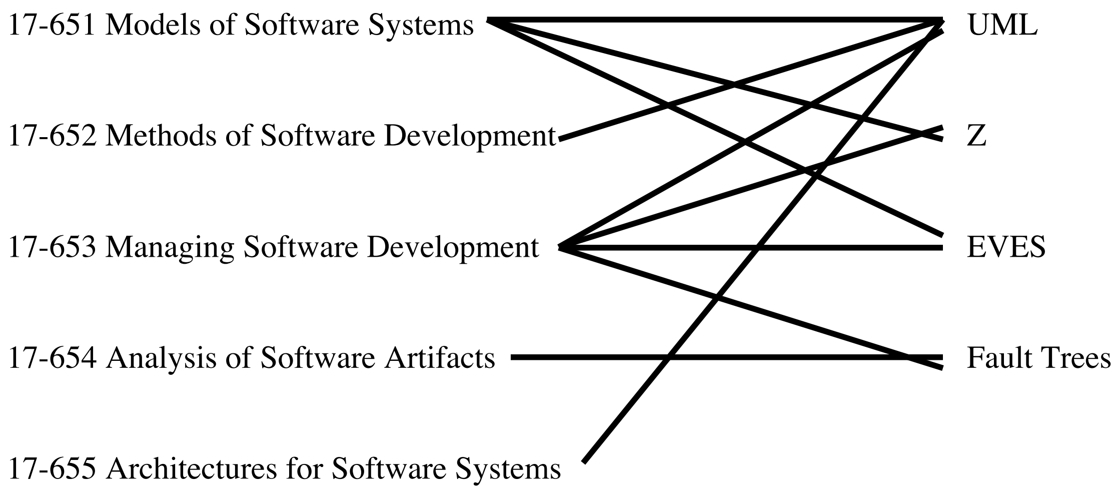
In all states of the vending machine system, the system should not dispense Coke or Pepsi when the temporary register contains fewer than seventy five cents.

If put through analysis software, the software will indicate whether this constraint holds for the system. If it holds, it means that the system will not allow a Coke or Pepsi to be dispensed when there are not sufficient funds in the temporary register. It will no indicate if an error has occurred, though violation of this constraint could indicate an error had occurred, and itself is constrainted by the specification of the system. If the system is specified incorrectly or incompletely, the analysis could be incorrect.

In the event of a constraint violation, a trace would be provided as to the order of events that led to the failure.

Appendix C: MSIT Core Courses

The following is a matrix that indicates which modeling techniques are used in which MSIT core curriculum courses.


17-651 Models of Software Systems
17-652 Methods of Software Development
17-653 Managing Software Development
17-654 Analysis of Software Artifacts
17-655 Architectures for Software Systems
UML
Z
EVES
Fault Trees