Recasting Risk Analysis Methods in Terms of Object-Oriented Modeling Techniques

Gregory D. Wyss, Ph.D.; Sandia National Laboratories; Albuquerque, New Mexico
Richard L. Craft; Sandia National Laboratories; Albuquerque, New Mexico
Ruthe L. Vandewart; Sandia National Laboratories; Albuquerque, New Mexico
Donald R. Funkhouser; Sandia National Laboratories; Albuquerque, New Mexico

Keywords: Risk assessment, object-oriented analysis, vulnerability assessment

Abstract

For more than two decades, risk analysts have relied on powerful logic-based models to perform their analyses. However, the applicability of these models has been limited because they can be complex and expensive to develop. Analysts must frequently start from scratch when analyzing a new (but similar) system because the understanding of “how the system works” exists only in the mind of the analyst and is only incompletely instantiated in the actual logic model.

This paper introduces the notion of using explicit object-oriented system models, such as those embodied in computer-aided software engineering (CASE) tools, to document the analyst’s understanding of the system and appropriately capture “how the system works.” It also shows that from these models, standard assessment products, such as fault trees and event trees, can be automatically derived.

Introduction

The term “risk assessment” is employed in many fields to describe a process for examining a system or process to determine whether it can be compromised by either natural or human actions, and whether such a compromise would lead to unacceptable consequences. The specifics of the risk assessment process, however, can vary widely from industry to industry and practitioner to practitioner. To some, risk assessment implies the use of formalized logic models such as fault trees and event trees. To others, it simply means an enumeration of threats and countermeasures. In any case, risk assessment methods generally examine a system from a “consequence-based” perspective in which the analyst first identifies the system-level consequences to be avoided and then proceeds to systematically decompose the events that cause those consequences into their fundamental causes. The analyst also seeks to understand the likelihood of each scenario that can lead to the undesired consequences, as well as the combinations of safeguards that might provide the best reduction in risk.

In order for a risk assessment to be “complete,” the list of undesired consequences must consider not only safety, but also security and reliability because, for example, a security “feature” may also present itself as a safety flaw or a reliability problem. Sandia National Laboratories refers to the simultaneous consideration of these design issues as “surety analysis.” The basic objective of surety analysis is to ensure that a system works properly when it is supposed to, and that it behaves gracefully under all plausible adverse conditions so that the incidence of undesired consequences of all types can be minimized.

Neither risk nor surety analyses can be performed without a sound understanding of the system being assessed. However, this fundamental facet of the surety analysis often exists only in the mind of the analyst. Assumptions about and misunderstanding of the system manifest themselves in incomplete and/or erroneous analyses, lack of reproducibility between analyses, and the time and expense that are involved in generating new analyses.

Since the Fall of 1996, Sandia has conducted a multidisciplinary internal research project aimed at creating an extensible framework capable of
DISCLAIMER

This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any agency thereof, nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof.
DISCLAIMER

 Portions of this document may be illegible electronic image products. Images are produced from the best available original document.
supporting a broad range of surety assessment techniques. The team is composed of members with backgrounds in computer security and information surety, probabilistic risk and reliability assessment, and object-oriented analysis methods. This team has come to believe that there is a fundamental concept that is common to all of these analysis methodologies which, if appropriately captured, would enable these paradigms to operate interchangeably from a common object model. Furthermore, once this object model is constructed, one can extract from it in an automated fashion many of the traditional risk, security, and surety analysis models, including fault trees, event trees, failure modes and effects analyses, as well as discrete simulations and vital area analyses. The object model instantiates in a formal way the analyst's understanding of the system because, through the use of object-oriented analysis techniques such as Shlaer-Mellor (ref. 1), the analyst encapsulates into the object model the behavior and causality that make up the system.

Overview of the Paper

The goal of this paper is to introduce the reader to the ways to use a common object model to perform a surety analysis of a "real" system. Such an analysis will make use of techniques that are derived from traditional risk and reliability analysis methods. Note that each such traditional method simply encapsulates certain aspects of the behavior and causality embodied in the system using its own particular syntax and logic rules. Therefore, if the system's behavior and causality were to be embodied in a properly constructed common object model, then one could, in theory, extract any type of risk or security model from the object model.1 This could be done automatically by interrogating the object model and translating the appropriate aspects of its contents into the syntax and logic rules required by the particular analysis method. In other words, once system behavior and causality have been embodied in the object model, the traditional risk and reliability analysis models can be had essentially for free because they are simply subsets of the object model translated into another syntax.

The common object model embodies the analyst's understanding of the system to be modeled. For this reason, the first section of this paper describes the ways that an analyst can look at a system to develop the understanding necessary to create a surety model. This is done in the context of five "views" of a system. The second section describes the "functional view" of the system. In this section, we describe most of the language elements that will be used for composing the object model for each of the five views. The next several sections describe the remaining four views and the mappings between them. Finally, we discuss the significance of using an explicit model of the system and describe how the model elements are used to support various traditional assessment techniques.

The Five Views

If one considers the entire collection of issues studied in a broad range of assessments, it becomes clear that understanding the system can entail investigating the system from a number of viewpoints2. These include understanding the system's purpose and behavior, its structure, its environment, the history of each of its components, and how each of these preceding four viewpoints can change over time.

The first view — the system's purpose and behavior — we refer to as the functional view of the system. The second — the system's structure — we call the physical view. The third is called the environmental view, and the fourth is called the life cycle view. The collection of information that specifies how each of these four views can change as a function of time we have named the temporal view.

The Functional View

The functional view documents what the system does without specifying how, where, or with which "real world" elements it does it. In this sense, the functional view is a logical view of the system. To capture this information, we have chosen to employ constructs from the object-oriented analysis techniques used in

---

1 According to Shlaer and Mellor (ref. 1), "An object is an abstraction of a set of real-world things such that all things in the set have the same characteristics, and all instances are subject to and conform to the same set of rules and policies."

2 Work done at the Queensland University of Technology (refs. 2-4) as well as examples from the qualitative reasoning community (ref. 5) have influenced much of our thinking on this point.
requirements engineering. In the functional view we address a number of questions:

- What are the blocks of functionality found in the system?
- What flows between these blocks?
- How do these blocks respond to the flows?

Sometimes it is useful to model a system as a collection of abstract entities. For example, in the remote monitoring system shown in Figure 1, the data processing center, the sensors, and the users can all be logical objects. While they may correspond in one-to-one fashion to tangible devices, they may not. In this example, the data processing center may be a single computer or it may be multiple networks of computers spread across multiple sites.

We refer to a drawing of the type shown in Figure 1 as a system structure diagram (SSD). Its function is to identify the logical components of a system and the communication paths between them. The data processing center and sensors are all logical components representing blocks of functionality present in the system. The user and the signal source components exist only to define the boundary of the system being assessed. The lines radiating out from the data processing center are communication paths, as are the lines between the signal sources and the sensors. Their function is to indicate the presence of flows between the components.

As part of modeling the abstract entities, we also identify flows between logical components. These flows represent the movement of information, material, or energy around the system. Figure 3 shows the SSD for the remote monitoring system with flows added. The seismic signals represent energy being transferred from the signal sources to the sensors. The sensor readings, sensor controls, and earthquake reports are all information flows. In our approach to modeling, a flow generated by a component terminates in one or more other components. In similar fashion, a component is capable of receiving a given type of flow from multiple components.

Once a flow has been identified in the system, we document its structure using combinations of attributes and components. For example, a sensor reading might be defined to include a date and time, a sensor ID, reading 1; reading 2; ...; reading 200, and a checksum. Each attribute assumes a value which is set by the component generating the flow. At the receiving component, the attributes will appear in a set of specifications that document how that component behaves.

If a flow contains components, the flow is interpreted in a different fashion. While the attributes in a flow can be considered "consumed" by the receiving component, components in a flow indicate dynamic relationships between the component(s) in the flow and those sending and receiving the flows. Figure 4 shows one example of this. A sterilization unit feeds empty pill bottles serially into the machine that fills them. Here, a given pill bottle initially has a relationship with the sterilization unit. However, as the flow occurs,
the bottle's relationship with the sterilization unit is terminated and a relationship with the filling unit is initiated. This is important because a central issue of risk assessment is access: which entities have the ability to interact with a given target. In this case, if one wishes to understand how the pills in a bottle can be contaminated, then it is important to know that the container holding the pills may be the vehicle for achieving contamination, and that the bottle had a relationship with the sterilization and filling units.

Everything described so far treats the logical components of a system as "black boxes." We have specified what goes into the boxes and what comes out, but we have not specified how the boxes transform inputs into outputs. To do this, we use state charts and data flow diagrams.

State charts capture the fact that a component may handle an input flow one way at one time and a different way at another time. We also use state charts to document a component's "macro-level behavior." For example, Figure 5 depicts the behavior of the data processing center shown in Figures 1 and 3.

![Figure 5 - Data processing center state chart](image)

The rounded rectangles are referred to as states. The arrows between the states are transitions. The labels on the arrow are events. States represent a body of behavior within the logical component. Events indicate when the component will stop performing one body of behavior and move to another. A transition indicates which body of behavior will become active given the current state and an event.

Events, in our modeling approach, come in five flavors. First, the arrival of an input flow to a logical component is an event. Depending on the component's current state, the component may or may not respond to this event. Second, completion of the "processing" in a state can qualify as an event. In this case, the event may cause the component to transition automatically to another state. Third, reaching a certain period in time can constitute an event. This period of time can be expressed in absolute terms (e.g., 11:59.59 p.m. Dec. 31, 1999) or in relative terms (e.g., 30 seconds later). Fourth, a given component attribute (or set of attributes) assuming a given value (or set of values) can be considered an event. An example of this is the overflow drain on a sink. With its drain closed, the sink operates in a given way as it is being filled; however, once the water level reaches a certain height, the overflow drain begins to operate. In modeling the sink, an attribute, "water level," could be used to track the current level of the water in the sink. When the level reaches a given threshold, the attribute could generate an event indicating to the sink component that its behavior is changing. Finally, we support random failures as events. This is to accommodate the notion of primary failures in system components.³

To document the behavior of a component in a given state, we use a data flow diagram (DFD). Figure 6 shows a DFD for the "generating earthquake report" state of Figure 5. In our modeling, DFDs are constructed from four building blocks: processes (e.g., "compiling report"), flows (e.g., "earthquake report"), stores (e.g., "users' database"), and attributes (e.g., "location" and "magnitude").

³ Analysts consider three kinds of failures: primary, secondary, and command. In a primary failure, the component in question fails on its own (e.g., as a result of aging). In a secondary failure, a component fails because it has been driven out of its design range (e.g., a pipe bursts due to excess pressure). In a command failure, a component operates as it should due to a valid command being sent to it, but this operation occurs at the wrong time (e.g., a wire in a circuit electrocutes a maintenance worker when someone resets the circuit breaker).
A process (denoted by a circle) can perform in various ways, including (ref. 1):

- transforming a collection of input attributes into output attributes
- testing a collection of attributes for a given condition, and
- generating events.

Flows are denoted using arrows and labels. The arrows indicate the flow’s origin and its termination(s). As with flows at the component level, a flow out of a process can have multiple termination points, and a given type of flow into a process can have several points of origination (signifying that any of these points could generate the flow). The labels on the flow represent either single attributes or collections of attributes. In Figure 6, the flows into “compiling report” are all simple attributes. The flow out, “earthquake report data” represents a collection of attributes. As in component-level flows, the structure of complex flows is defined. Unlike component-level flows, components are not included in the definition of the structure.

Two other flows worth noting in Figure 6 are the event flow, “earthquake report sent”, and the “bare” input flow into “compiling report”. The event flow corresponds exactly to the event of the same name on the state chart shown in Figure 5. When the “sending report” process generates this event, the state chart transitions to the “queueing sensor reports” state. The bare flow on the input of “compiling report” is typically referred to as a control flow. Its function is to “trigger” the “compiling report” process. In a DFD, a given process does not begin to operate until all of its inputs are present. Input flows from stores (such as the “earthquakes database”) are considered to always be present. In Figure 6, the control flow becomes active once its associated state is entered. This causes “compiling report” to execute and create the flow “earthquake report data.” Given that the flow “user e-mail addresses” is always available to “sending report,” this second process is executed to generate the “earthquake report” flow and the “earthquake report sent” event. This control flow can be used to indicate that nothing more than entering the state is required to start the first process. Note that control flows can also be used between processes to enforce a given order of execution (i.e., a process receiving a control flow will not trigger it until the process generating the flow has sent it).

Flows in a DFD can start and end in a number of places. As shown in the DFD, they can have their origins in stores or processes or can be rooted outside the diagram. In the latter case, this can represent an event that is generated by another state in the component or a flow received by the component from another component. Similarly, flows can terminate in processes, stores, or outside of the DFD. When they terminate outside of the DFD, they can represent output flows that the component sends to other components or events that drive state charts within the component.

The final feature to note in the DFD is the store. Stores represent memory in the component that contains them. The goal here is to model the fact that a component can collect “stuff” which can either affect its subsequent performance or be consumed at a later time. For example, a component representing a tank might employ attributes to track the temperature and composition of its contents. Since these contents might be quite complex, the analyst may choose to create a group of attributes in the component attribute called “contents.” This group could then contain one or more records defined as “element + quantity”.

While it is typical in object-oriented analysis to represent this memory as a simple collection of attributes, we have chosen to allow related attributes within the component to be grouped into named collections. The primary reason for this is that an analyst, while choosing to defer the decomposition of a complex component, may still want to track the existence of collections of attributes that are important to understanding the component’s operation. When the component is decomposed, these collections become the attributes associated with the constituent components.
Before ending this discussion on modeling the logical view of a system, we must note that we permit a component to contain multiple state charts. The intent is to capture the fact that a component can exhibit multiple simultaneous behaviors. A PC is an example of this. While it is processing, it is also producing heat and consuming power. If we chose, we could create state charts for all three phenomena. Since these models are clearly not independent (e.g., loss of power causes processing to stop), we also allow for two forms of communication between state charts. In the first, a process in one state chart alters an attribute that is read by a process in another state chart. In the second, a process in one state chart generates an event that triggers a state transition in another state chart.

The Physical View

Whereas the functional view of the system documents what the system does, the physical view documents what the system is. Each of the real-world components that delivers some portion of the functionality is identified in the physical view, along with specifications for how these components relate to one another.

Figure 7 shows a schematic for the “data processing center” in Figure 1. Just as the logical view shown in Figure 1 says nothing about how the system is implemented, Figure 7 says nothing about what the system is used for. This configuration of communications and computers could just as easily be for an on-line gaming system as it could be for the analysis and reporting of earthquakes. Clearly, this view addresses the “identify the assets” step found in most methodologies. As such, the elements that populate this view need not be just hardware. They can also be software, people, pipes and pumps, or other real-world entities.

As in the functional view, components in schematics can be decomposed into their constituent components. Figure 8 does this for the Sparc20. Both here and in Figure 7, the boxes represent physical components, and the lines indicate communication paths between these components. In Figure 8, flows have been added like those in Figure 3. Their meaning is the same as before: they represent communications between components in this system. Here, the “status” message can take on two values ("enabled" or "disabled") in order to provide a mechanism for the processor board and operating assembly component to tell the software components whether they are functioning (for example, if the processor ceases to function due to lack of power).

Mapping the Functional to the Physical View

While the functional and physical views of the system are different, they are not independent. Rather, components and flows in the functional view exist in the context of components in the physical view. To document this fact, we develop a map between the functional and physical views as shown in Figure 9.
implied relationship here is that a given physical component supports a functional component (i.e., failure of the physical component results in failure of the logical component). However, other more subtle relationships are also possible.

One might ask, “Why is this mapping needed?” There are at least two reasons. First, it is possible that the system to be assessed has no physical reality. For instance, a system that is in the architectural phase of its design may only be defined as a collection of behaviors, and these behaviors may have yet to be assigned to physical components. Second, in some systems, functional and physical component are not synonymous. For example, in some computing environments, processes that embody a given function are shuffled between computers as needed to balance workloads on the computers. In addition, adaptive command-and-control structures (structures that enable a system to adapt to decapitation and other situations where one entity is unable to carry out its mission) often exhibit similar behavior.

The Environmental View

Just as functional components reside in physical components, physical components exist in their own context. This context is documented using the environmental view. While this view can be divorced from a reality (for instance, the environmental view might be composed entirely of named places, such as building 32, that are not defined in terms of a coordinate system), this view is typically used to document the spatial characteristics of the system.

Defining the environmental view involves two main steps:

- Naming regions in the space occupied by the system (a region typically represents an area subject to a common set of influences, such as temperature, humidity, water, fire, or access/invasion), and
- specifying how physical elements in the system and environment relate to each other in space

While the first task is purely logical in nature, the second introduces a new feature — a coordinate system — to the tool.

Mapping the Physical to the Environmental View

Once the components that populate the system’s environment have been defined, the analyst can specify the relationship between physical and environmental components. This relationship can assume two forms. In the first, the analyst simply specifies which physical components are found in which environmental components (i.e., named regions). In the second, the analyst is more specific, using physical coordinates to say exactly where in space each physical and environmental component exists. Note that this can be done in either absolute or relative terms. Relative terms can be used to specify how assemblies relate to each other. Without stating exactly where the assembly exists in the system space, the analyst can specify where the elements of the assembly sit relative to one another. By contrast, when absolute terms are used, the analyst specifies exactly where in the model’s space each component or assembly is found and how it is oriented at that location.

The Temporal View

So far everything described in this paper has assumed an essentially static view of the world. The problem with this is that many systems are not static. One time dependency issue is that the system configuration itself changes over time. Components come and go, and flows that exist at one moment are impossible at the next. Our current best approach, which is less than ideal, is to treat each change in system configuration as a new system and assess it separately.

A second temporal issue for systems is timing (as opposed to time). Timing issues can, for example, determine whether a given scenario is even possible. Our approach to timing analyses is to assign propagation delays to the processes in the system components. These delays are then used to prune the fault space to determine whether any impossible fault sequences can be found. Similarly, normal system operations are assessed to determine whether timing issues can force undesired outcomes.

The Life-cycle View

The fifth and final view of the system is related to the temporal view: it is the life-cycle view. This view is used to capture the fact that at times problems are introduced into a system by events.
that affected a component before that component ever became part of the system. An adversary might be able to subvert a system at a number of points in its life cycle. These could include design, implementation, transportation to the field, installation, or maintenance. To address this fact, we use life-cycle diagrams to document the life cycles of components. These diagrams are similar to the state charts in which the states represent points in a component’s life cycle and the transitions represent the way the component moves from stage to stage in its life cycle.

The significance of a stage in the life cycle is that in that stage the component exists in a different configuration. Since the life cycle can be documented for any component in a system, the collection of life-cycle diagrams can be thought of as a network of systems to be assessed. Systems that exist “upstream” in the network have the potential to affect systems “downstream” by altering the state of components before they reach their primary system configuration.

Significance of this Approach

In risk assessment, one of the key issues is the question of causality — what could have happened to cause a given condition to occur. In practice, an “undesired outcome” is rarely the result of a single event. It is more common for a chain (or chains) of events to lead to the outcome. Standard approaches for documenting these chains of relationships include event trees and fault trees.

In current practice, constructs like fault trees are “hand-tooled” by the analyst. This approach has several shortcomings. First, while the elements of the tree are rooted in the analyst’s understanding of the system being assessed, nothing in the trees states explicitly what the analyst understands about the system. As a consequence, assessment results are often highly dependent on the analyst conducting the assessment — different analyst, different results. How the analyst derived a tree is often unclear. Whether facts were excluded from the tree because the analyst considered them unimportant or because the analyst failed to understand that part of the system may not be known to someone reading the tree.

Second, effective use of tools like fault trees and event trees requires that the analyst take small conceptual steps and not grand leaps of faith regarding cause and effect. For instance, it is better to state that:

- if the power fails, the inmates might escape
- if an inmate escapes, he might commit additional crimes
- if an escaped inmate commits a crime, then the warden might be fired
- if the warden is fired for mismanagement, he is likely to be hired as ...

than to state:

- if the power fails, then the warden is likely to be hired as ...

While this relationship may be true, its logic is not as traceable as that of the first set of assumptions. Grand leaps of faith make verification of the logic difficult or impossible.

Third, a given initiating event might have additional side effects that are important to understand. For instance, in a battlefield environment, a commander may want to deprive the enemy of the use of a given communication link. The logic is something like this:

- I destroy the link, therefore the enemy cannot communicate orders to the troops
- If orders do not get to the troops, then they will not respond effectively to our attacks
- If they do not respond effectively to our attacks, we will win.

At the same time, there may be another equally valid, yet unseen logic at work:

- I destroy the link, therefore the enemy cannot communicate with the troops, I lose an important intelligence source
- If I lose this intelligence source, I no longer have effective insight into enemy plans
- If I lack this insight, I may not be able to plan my attacks as effectively
- If my attacks are ineffective, we may not win.

When implicit models (the knowledge in the analyst’s head) instead of explicit models drive the development of assessment logic, it is easier to miss important logic chains. Explicit models encourage completeness in the risk assessment process.

How the Models Are Used

While we have only discussed our modeling approach in the context of the functional view of the system, the same basic constructs can be used to model the system’s physical structure,
the components that populate the system’s environment, and the abnormal or fault behavior of each component in the system. When a system is modeled in this way, the causal relationships needed to define event trees, fault trees, and other assessment structures are already present in the model. Development of these structures is then relatively straightforward.

For instance, to develop a fault tree, the analyst might proceed as follows:

1. Select an attribute in a component or flow and pick an attribute value that represents an undesired outcome. For example, in Figure 3, select the “earthquake report” flow and specify that the undesired outcome is an alteration of the recipient’s e-mail address in the flow. This becomes the top-level event.

2. Starting with the point at which the undesired outcome is realized, begin tracing backward through the chain of flows and components to determine how the undesired outcome might be realized.

3. Since the starting point is a flow, the analyst assumes that one of two conditions could result in the alteration of the e-mail address: either it was changed in the flow itself or it was changed prior to ever entering the flow. At this point, the fault tree has grown from a top-level event to include an “OR gate” that is fed by two events: “address corrupted in flow” and “address corrupted in data processing center.”

4. While the first of these two events remains undeveloped, development on the second begins by tracing the flow backward into the data processing center component. As noted earlier, an output flow from a component is really the output flow from one or more processes in the component. If “earthquake report” were generated by multiple processes, then an “OR gate” feeding into the event, “address corrupted in data processing center”, would be added to the fault tree to account for the fact that the corrupted attribute could have been generated in multiple locations. In fact, only one process generates this flow, so the tracing of the flow now goes as far back as the output of the process, “sending report.”

5. The way in which the analyst traces through a process depends on the type of process. In this case, the process simply aggregates information presented on its inputs. In effect, the process is a conduit. As with the output flow where this backtracking began, the analyst realizes that a corrupted address could come out of the process for one of two reasons: the process somehow corrupted it, or the address that reached the process was already corrupted. These two possibilities cause an OR gate feeding “address corrupted in data processing center” to be added to the fault tree with two events feeding the gate: “address corrupted by process ‘sending report’” and “user e-mail addresses corrupted.”

6. Since the “sending report” process is logical, the ways by which it could fail or be subverted are not known. Thus, the associated failure event remains undeveloped. On the other hand, the “user e-mail addresses corrupted” event represents the state of the flow and can be developed in the same way as the previous flow. This results in an OR gate feeding this event with “addresses corrupted in flow” and “addresses corrupted in user database store” feeding the gate.

This process of backtracking continues until every branch of the fault tree reaches the system boundary or terminates in an undeveloped event.

These steps have described an analyst tracing through the system model to produce the fault tree. However, this need not be the case. The rules for traversing the graph and picking out events are straightforward enough to permit automation. Just as the model base can be used to develop deductive structures like fault trees, it can also be used to automatically build inductive structures like event trees. Also note that, once the system model is built, the analyst can quickly generate a number of different assessments with very little effort.

At times, mechanisms other than logic structures are useful for assessment. For instance, heuristic graph searching techniques are useful for modeling active adversaries. Similarly, simulations are a useful mechanism for gaining insight into a system’s operation. We believe that the system models developed using this
approach will be equally useful for these assessment methods.

Conclusions

"Understanding the system" is a critical first step in risk assessment. We have shown that by explicitly documenting the analyst's understanding of the system, other products of analysis, such as fault trees and event trees are obtained automatically. In addition, making the model explicit makes the results of assessment traceable and more complete.

Bibliography


Biographies

Gregory D. Wyss, Ph.D., Principal Member of Technical Staff, Risk Assessment and Systems Modeling Dept., Sandia National Laboratories, PO Box 5800, MS 0747, Albuquerque, New Mexico, 87185-0747, USA, telephone - (505) 844-5893, e-mail - gdwyss@sandia.gov.

Dr. Wyss has developed new methods for probabilistic risk assessment (PRA) and uncertainty analysis for more than 11 years. He has performed these studies for systems ranging from telecommunications networks to nuclear power plants and the Cassini space probe.

Richard L. Craft, Senior Member of Technical Staff, Distributed Systems Assurance Dept., Sandia National Laboratories, PO Box 5800, MS 0455, Albuquerque, New Mexico, 87185-0455, USA, telephone - (505) 844-8873, e-mail - rlcraft@sandia.gov.

Mr. Craft has spent the past four years as an information systems security analyst. He also has more than ten years of systems analysis experience using object-oriented techniques. His assessments have included a number of national security and command and control systems.

Ruthe L. Vandewart, Principal Member of Technical Staff, Decision Support Systems Architecture Dept., Sandia National Laboratories, PO Box 5800, MS 1138, Albuquerque, New Mexico, 87185-1138, USA.

Ms. Vandewart designs and implements object database management systems and object-oriented software architectures.

Donald R. Funkhouser, Principal Member of Technical Staff, Decision Support Systems Architecture Dept., Sandia National Laboratories, PO Box 5800, MS 1138, Albuquerque, New Mexico, 87185-1138, USA.

Mr. Funkhouser specializes in the design and implementation of object-oriented software architectures using Java and other languages.