Software Engineering Institute | Carnegie Mellon University

General Modeling Concepts

Samuel T. Redwine

February 2007

ABSTRACT: This document introduces several concepts and issues that provide a background for understanding the Introduction to Modeling Tools for Software Security article and modeling in general.

- The subjects covered are
- Structure of a Model
- Purpose and Scope of Model
- Level of Abstraction
- Level of Rigor
    - Informal
    - Semiformal
    - Formal
- Logic: Predicate and Temporal
- State Machines and Model Checking
- Solution Under Constraints
- Summary

Notations mentioned and/or characterized include informal graphical and textual ones such as English prose, semiformal (structured), and formal (mathematically based).

STRUCTURE OF A MODEL

Models represent some aspects of the items being modeled and may be used for a variety of purposes. Multiple models may exist to represent the same or different aspects, raising issues such as clean separation of concerns and comparison and composition of multiple models. Models may be static (such as a one-piece figurine, a diagram, an equation, or a logical expression) or dynamic (such as a model electric train or a computer simulation). Executable models generally take input and produce output using some method that has a basis relevant to the situation. The modeling method may incorporate systems, software, security, or application domain knowledge or be a general one.

General modeling methods range from mathematical logic, regression analysis for trend lines, and statistical techniques through general models of information, flows, queues, grammars, and state machines to specialized methods for security,

Software Engineering Institute
Carnegie Mellon University
4500 Fifth Avenue
Pittsburgh, PA 15213-2612

Phone: 412-268-5800
Toll-free: 1-888-201-4479

www.sei.cmu.edu

software applications, or particular programming languages. Thus, you can think of models as having a conceptual basis, such as state machines or predicate logic, and embedded knowledge related to the area modeled. The amount of information about the conceptual basis or embedded knowledge that is visible to the user varies with some modeling tools, requiring users to have no knowledge of them.

Thus, execution of a model involves initial and possibly ongoing input and embedded knowledge that, along with the input, is manipulated using some conceptual basis to achieve single or multiple values of final or intermediate results useful for exploration, analysis, or prediction.

The structure of the model may reflect the structure of the software design, say in a state machine, or not, say in a statistical prediction based on counting objects or lines of source code. The modeling may be entirely concerned with static descriptions or artifacts or may include dynamic models and analyses involving simulated or actual execution.

PURPOSE AND SCOPE OF MODEL

Do you want to know if your specifications conform to the required security policy? Want to know what would happen if you designed something a certain way? Does your design have questionable features? Does it meet the specifications? The security policy? Models and modeling tools may be able to help you answer these questions.

In creating, modifying, and judging systems, one might use models to record, explore or achieve insight, analyze, or predict regarding artifacts or situations. Typically, the model is an abstraction of the specification or design covering only certain aspects but not others, such as functionality but not timing, or concurrent but not (irrelevant) sequential behavior. While most modeling of specifications and designs relates to some aspect of execution behavior, the model could be about any portion of the life cycle—developmental, operational, or otherwise—and could address the system's or product line's environment as well as the system itself. The model may concern itself with the software, its environment, or both and their interactions. It may be normative, resulting in statements of compliance, correctness, or merit, or it may state results without judging them.

Usually in creating, modifying, documenting, and judging systems, one models for one of six direct reasons:

1. exploration and the seeking of insight (e.g., using simulation)

2. showing the internal consistency of a specification or design
3. showing that the software is correct or at least compliant with the security policy (e.g., via formal methods using mathematical logic and proofs)
4. discovering faults or weaknesses (e.g., modeling designs and possible attack patterns)
5. achieving measures of merit, including comparing the merits of multiple designs
6. aiding some non-design activity such as testing (e.g., state machine models of designs and enumeration of the states and paths for generating test cases [<a data-cke-saved-href="/articles/tools/modeling-tools/modeling-tools-references" href="/articles/tools/modeling-tools/modeling-tools-references" blackburn2001"="">Blackburn 2001, Sinha 2006])

This last might be a breadth first enumeration of states the design might reach during execution as well as example paths to these states thereby suggesting states and paths to be tested for conformance of the executable with the design. Other means of simulation and design by contract can also be used to aid testing.

Scope can vary in parts and aspects of the system included. In describing a specification or design, the following are typical elements of the description:

- Views
    - Static views – logical, structural, location in context, etc.
    - Dynamic views – changes of behavior
    - Constraint specifications – constraints requiring compliance
- Consistency specifications – requirements for consistency within or among parts of the description (or possibly with actual dynamic behavior)
- Measures of merit
- Relationships
    - Instantiation – A2 is a specific instance of “class/type” A1.
    - Refinement – A2 represents a more detailed or concrete description of A1.
    - Realization – A2 represents an implementation of A1.
    - Specialization – A2 is a specific instance of “generic” A1.
    - Uses – A1 depends on A2 in order to behave properly.
    - Rationale for – X1’s rationale is given in A2.
    - Derived – A2 is a logical consequence of, and generated from, A1 (traces from).
- Means of creation or modification
    - Generated – created by automated actions
    - Implied – deduced by applying a set of rules
    - Manual – created by the actions of a human being

Normally, models would not include all of these but only those needed or relevant.

Because the time taken by the modeler or the modeling tool expands with a model's scope or size, modelers face the problem of determining the scope or size of a model needed to ensure the validity or adequacy of the results from using the model for a particular purpose. This might involve choosing certain parts of the system to include in the model or another dimension such as the number of execution alternatives (e.g., number of similar concurrent processes) modeled.

The purpose and scope of the questions you are attempting to answer by modeling determine the relevance to you of various models and modeling techniques. This may be straightforward when the model or technique was created or is regularly used for addressing the kind of question of interest to you. If no such model or technique exists or is satisfactory, then judging suitability may require special modeling expertise and experimentation or adaptation.

LEVEL OF ABSTRACTION

A model may be more abstract than a representation, behavior, or other model if it concerns itself with less. It may have fewer aspects or details. Examples include timing but not functionality, fewer states within a component or values for a variable, and what but not how. Typically, it is less primitive and further from being executable. By definition it is less concrete, as "to make more concrete" is the opposite of "to make more abstract."

Programming languages such as Java are more abstract than assembly language. The Java Virtual Machine is said to be an abstraction of a computer because it is not a concrete computer and lacks many of the aspects of hardware, such as weight (although some examples of Java machine chips have been created).

We often speak of levels of abstraction in representing a system, with specifications being more abstract than high-level design, and so on through detailed design, code, and the binary executable (see "Typical Levels of Abstraction for System Descriptions" below). Here the essence of a representation being at a higher level of abstraction is that the behavior and/or the details of the software are less determined—that is, constrained or established—than at the next lower level. This means that additional design decisions lie between a higher level representation and next lower one, with the lower level one stating more about how things will be done. A human may make these decisions during development, using a tool such as a source code generator or a compiler, or by input at runtime.

Typical Levels of Abstraction for System Descriptions

- Specification of Externally Visible Behavior
- High-Level Design
- Detailed Design
- Source Code
- Object Code
- Binary Stored and Loaded onto Computer
- Binary Executing in Machine

The merits of the alternative chosen for how to create a description and the consistency of the lower level representations with higher ones (its correctness) are the subject of much of design modeling. A model may address this question of consistency or correctness for only limited aspects of the system. For example, one might sometimes be concerned only with meeting security constraints on behavior, such as the confidentiality of the response, but not whether the response has the right value.[1]

As with increased scope, the lower the level of abstraction, the greater the time taken by the modeler or the modeling tool, reflecting the increase in detail. Thus, when using a model for a particular purpose, modelers must select the level of abstraction for their model and how much design detail is necessary to ensure the validity or adequacy of the results without making the model too large for understandability and review or for the tool used and computational resources available.

The same project may use several models of the software's or system's design or design alternatives at different levels of abstraction as it evolves from high-level design through detailed designs. Possibly, as with analyses of source and binary code, these may seek to avoid successful "attacks beneath the level of abstraction" of your analyses or verification.

LEVEL OF RIGOR

[1] Sam Redwine has suggested that wanting the answer to be confidential when it is wrong has a place in user considerations.

Specifications and descriptions of designs can use notations that range from an informal natural language such as English or diagrams whose symbols have no explicitly defined meaning through structured descriptions using tables or diagrams with informally defined semantics to formal, mathematically based notations. The ability to reason about the design and the power of models tends to increase the higher the level of formality (i.e., increased rigor) of their description. This section explores this range of rigor.

Informal Descriptions

The first descriptions of a design are often informal sketches or in native language such as English.[2] Informality has its merits. In addition to sometimes being the easiest way to initially capture facts or thoughts, natural languages have the advantage that many kinds of stakeholders can read and review them. As everyone has experienced, pictures or informal diagrams can help one understand ideas, despite their lack of rigor.

However, difficulties arise when you attempt to do real engineering. Unfortunately, if you stop with an informal description, your only description can suffer from difficulty of understanding or even meaninglessness, ambiguity, inability to be adequately reasoned about or analyzed, or poor writing skills. For example, it may be excessively brief or verbose.

While writing things down is generally better than not doing so and can significantly aid understanding, informality also allows different people to read different meanings into the same words or graphics. In addition, informal descriptions can lose meaning with the passage of time. (What did we mean when we said that?)

Semiformal Descriptions and Notations

Not surprisingly given this, good designers tend to move to organized or structured descriptions with better-defined semantics. Lists and tables and glossaries or controlled, defined vocabularies can be significant steps forward. You have probably experienced the advantages of transforming requirements, say needs or features, into lists of separated individual requirements, each with a permanent ID. Communication, traceability, decision making, and sizing all become easier to do.

[2] An old engineering joke is that great ideas are first sketched on a napkin or placemat.

Informal diagrammatic notations such as class diagrams in UML, which lack formal semantics, are nevertheless useful for creating, recording, communicating, reviewing, and improving aspects of design. Dataflow diagrams can help one visualize paths for security attacks on a design.

Despite their great usefulness to designers for some tasks, informal and semiformal notations have serious shortcomings when it comes to modeling security properties of designs.

- Ambiguities or incompleteness can result in
    - mistakes in validating the specifications and security policy suitability or correctness
    - missing internal inconsistencies
    - mistakes or uncertainties in the verification of a design's consistency with specifications or security policy
    - implementations of the design that do not agree with what the designer meant when he or she wrote the design down
    - misunderstandings by testers, leading to improper tests
- The inability to do automated analyses can result in
    - overlooking faults or weaknesses that analysis could have caught
    - difficulties in reasoning or modeling
    - greater levels of uncertainty regarding compliance with the system's security policy and correctness in general

One result of these problems is the need for substantial and often especially skilled manual effort, not just in aspects of design such as design verification but elsewhere such as in reviewing source code and testing. Not to mention that persons highly skilled in designing for security and doing security reviews of designs are currently rare and expensive. These same characteristics of informal and semiformal descriptions also raise significant difficulties in modeling designs and ensuring security properties.

However, when you are seriously concerned about security, one design problem stands out above all others. Because security weaknesses frequently lie at obscure places or in rare or unheard of events, humans relying on these notations have a tendency to overlook them. Thus, methods that cover all possibilities are necessary to expose them and ensure they are handled properly. In real systems, covering all possibilities via testing is impossible, so other methods that rely on formality and modeling tools are needed.

Using much of UML for modeling faces this problem. A number of approaches have been taken to make it more suitable for addressing security or making it more formal, including by Jürjens ([Jürjens 2005] and [Jürjens 2004]) and others

([Fan 2006], [Funes 2002]. Likewise, aspect-oriented approaches span levels of formality, including programming languages, which are, of course, formal.

Formal Descriptions and Notations

Numerous formal notations exist, but only a few are in relatively wide use. This discussion characterizes and explains advantages and disadvantages of several approaches at a high level. Many Web links exist in the formal methods virtual library, particularly for tools.

Logic: Predicate and Temporal

Everyone in computing has been exposed to a logic notation, even if only in the condition within If statements. Logics are concerned with reasoning about the truth or falsity of statements. Propositional and predicate logics go from simple ones that have And, Or, and parentheses and ones that add For Every and There Exists to ones that have predicate statements or functions as arguments in other functions. These are some of the most notable notations:

- Z: a specification and design notation based on simple discrete mathematics—predicate logic, sets, functions, relations, and arithmetic—that allows modular description of state and operations plus their composition of multiple operations [Spivey 1992]
- VDM: a specification and design language that shares many of the features of Z but can also explicitly deal with undefinedness
- B: also shares many similarities, but emphasizes abstract machines and has an accompanying programming language usually associated with an automated toolkit
- ACL2: a specification and programming language combination explicitly backed by an automatic proof tool
- PVS: a notation associated with a powerful automatic proof tool that, because of the tool’s power, is also chosen by some for describing aspects of the specification and design

While the last three are explicitly associated with tools, the first two also have tools available to support them and many articles describing their use in research and practice. All but the last have multiple books dedicated to learning and using them. Z has the largest body of industrial use, but the others have also been used in real projects.

Although one can reason about a variable called time, predicate logics do not have an inherent sense of time. However, temporal logic does with additional operators such as Henceforth, Eventually, Until, and Next. These can be particularly useful when modeling concurrency.

State Machines and Model Checking

Finite state machines with their states and operations that transition between those states are used in many places in computing and represent a powerful model and, with rigorous semantics, a formal one as well. They can be used in design descriptions, simulations, test generation, and elsewhere either as single machines or in combinations of machines. The most common automated analysis of state machines and combinations of state machines is to identify all the states the system can ever reach. This is called reachability analysis or model checking.

Model checking tools have become surprisingly powerful and are able to handle enormous numbers of states. Commonly used tools include SVM and SPIN. The most common uses for model checking are concurrency and protocol analyses. Concurrency is a difficult and mistake-prone area of design, and its automated analysis has proven useful. Despite the huge number of states that modern tools can handle efficiently, abstraction of a design to have fewer states is often needed. Even so, analyses are not always carried to conclusion. Rather, a breadth-first exploration of reachable states might be done to identify all states that can be reached from its starting or initial state in, say, ten operations.

SVM and SPIN have native notations—PROMELA in the case of SPIN. However, they have often been used to do model checking on designs specified in other notations.[3]

Solution Under Constraints

Often called by the misleading title of "mathematical programming," a set of modeling and solution or optimization techniques exist such as linear programming (constraints and value to be optimized are linear) and integer programming (uses integer values). One security-oriented use of such techniques is described in [Wagner 2000], where an integer constraint modeling approach is taken to identifying buffer overflows in C.

Bases for Reasoning

Models may be created at different levels of abstraction and rigor using a variety of notations. These support a variety of bases for reasoning and calculation about a system and its attributes (see "Some Bases for Reasoning" below).

[3] Another kind of notation where model checking is used that might not appear on the surface to be a natural fit are process algebras, which are not described here. For the FDR2 model checker and other tools for the most widely used process algebra, CSP, refer to the FormalSystems website [FormalSystems 2008].

Some Bases for Reasoning

- Quantitative
    - Deterministic
        - E.g., predicate logic proofs, deterministic finite state machines
    - Non-deterministic formal systems for reasoning
        - Probability
            - E.g., non-malicious risk, Bayesian networks
        - Game theory
            - E.g., minimax
        - Other uncertainty-based formal systems of reasoning
            - E.g., fuzzy sets, others used in AI
- Qualitative
    - E.g., process maturity, staff skill and experience, compliance with standard, qualitative statements of event causality, subjective judgments

These bases have differing power and provide results with different meanings. Thus, the usefulness of each varies with your purpose for modeling, the nature and size of what you are modeling, how much you know about it, and the level of uncertainty you desire in your results.

SUMMARY

Your situations, needs, and purposes vary over time. Therefore, you may gain from creating appropriate models at different levels of abstraction and rigor using differing notations.

You should give careful thought to the purpose and scope of your model or modeling effort. Different modeling approaches may be more useful for exploration or verification, or finding mistakes versus assuring compliance or correctness. A model's internal structure may or may not reflect the structure of the design. Important attributes are the level of confidence you can justify in the model and the adequacy of its output for the purposes for which you wish to use it.

Because security weaknesses frequently lie in obscure places or in rare or unheard of events, humans have a tendency to overlook them. While tools that specialize in finding certain kinds of faults or weaknesses may discover some, methods that cover all possibilities are necessary to expose all of them and ensure they are handled properly. In real systems, covering all possibilities via testing is impossible, so if this is your goal, as well as for many more limited scopes or purposes, useful methods are available to you utilizing modeling tools and a range of formality.