Formal Methods

Chapter 21
Outline
Formal Verification Techniques

• Formal specification languages for specifying requirements and systems
  • Well-defined semantics, syntax
  • Based on mathematical logic systems

• Mathematically-based automated formal methods for proving properties of specifications and programs
  • Inductive verification techniques
  • Model checking techniques
Inductive Verification vs. Model Checking

Classification criteria:

• **Proof-based vs. model-based techniques:**
  - *premises* embody system description
  - *conclusion* represents properties to be proved
  - Proof-based: derive intermediate formulae that go from premises to conclusion
  - Model-based: establish that premises, conclusion have same truth table values

• **Degree of automation**: fully manual to fully automatic, with everything in between
Inductive Verification vs. Model Checking

Classification criteria:

• **Full vs. property verification:**
  • System specification may describe entire system or part of system
  • Property specification may be single property or many properties

• **Predevelopment vs. postdevelopment:** may be design aid or for verification after system design is complete

• **Intended domain of application:** hardware or software, sequential or concurrent, non-terminating (like an operating system) or terminating, and so forth
Example: HDM

• Developed at SRI

• Began as proof-based formal verification methodology
  • Covers design through implementation
  • Automated, general-purpose methodology
  • Used specification languages, implementation languages

• Provided model checking with its multilevel security tool
  • Input is formal specification in language SPECIAL
  • Theorem prover uses proof-based technique; fully automated property-oriented verification system
Example: HDM

• Tool uses SRI model (interpretation of Bell-LaPadula model)
  • Given a SPECIAL specification
  • Verification condition generator creates formulae that assert specification correctly implements SRI model
  • Boyer-Moore theorem prover processes these formulae
  • Output is list of the formulae that were satisfied and those that were not
Formal Specification

• A specification written in a formal language with restricted syntax, well-defined semantics, based on well-established mathematical concepts
  • Precise semantics avoids ambiguity
  • Languages support exact descriptions of system function behavior
  • Generally eliminate implementation details
• Automated tools support verification of syntax, semantics
Example Language: SPECIAL

• First-order logic-based language
  • Nonprocedural, strongly typed

• Specification in SPECIAL represents module
  • Specifier defines module scope
  • Systems described in terms of modules

• Function representation in modules
  • VFUN: describe variable data
  • OFUN: describe state transitions
  • OVFUN: describe state transitions and changes in VFUN values
Bell-LaPadula Model and SPECIAL

**MODULE** Bell_LaPadula_Model give-access

**TYPES**

Subject_ID: DESIGNATOR;

Object_ID: DESIGNATOR;

Access_Mode: {OBSERVE_ONLY, ALTER_ONLY, OBSERVE_AND_ALTER};

Access: STRUCT_OF( Subject_ID subject;

Object_ID object;

Access_Mode mode);
Comments

- Subject_ID, Object_ID types described at lower level of abstraction
  - The DESIGNATOR indicates this
- Access_Mode types have 3 possible values
- Access type is structure with 3 fields of types shown
Bell-LaPadula Model and SPECIAL

FUNCTIONS

VFUN active (Object_ID object) -> BOOLEAN active:
HIDDEN;
INITIALLY
    TRUE;

VFUN access_matrix () -> Access accesses:
HIDDEN;
INITIALLY
    FORALL Access a: a INSET accesses => active(a.object);
Comments

• VFUN $active(object)$ defines the state variable $active$ for the $object$ and sets it to $TRUE$ initially
  • So state variable for that object is true if the object exists
• VFUN $access_matrix()$ defines the state variable $access_matrix$ to be set of triples ($subject$, $object$, $right$)
  • This is simply the current set of access rights in the system
Bell-LaPadula Model and SPECIAL

OFUN give-access(Subject_ID giver; Access access);

ASSERTIONS

active(access.object) = TRUE;

EFFECTS

access_matrix() = access_matrix() UNION (access);

END_MODULE
Comments

• OFUN *access_matrix()* defines state transition when new object added to matrix

• State variable *active* for object must be true
  • See in the **ASSERTIONS** sections

• Value of state variable *access_matrix* after transition is value before transition and additional access rights for the new object
Hierarchical Development Methodology (HDM)

- General-purpose methodology for design, implementation
  - Goal was to automate and formalize development process
- System design specification is hierarchy of a series of abstract machines at increasing level of detail

```
<table>
<thead>
<tr>
<th>Requirements</th>
<th>Analyze, accept requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>Model proven internally consistent, used as basis for verifying lower AMs</td>
</tr>
<tr>
<td>External Interfaces AM 1</td>
<td>First AM is usually external interface, called Formal Top Level Specification</td>
</tr>
<tr>
<td>Abstract Machine AM 1</td>
<td>Each AM mapped to next lower AM, which represents lower levels of system specification</td>
</tr>
<tr>
<td>Primitive Machine AM n</td>
<td>Some combination of hardware and software that runs verified system</td>
</tr>
</tbody>
</table>
```
Specifications

- **Hierarchical** specification identifies abstract machines (AMs) making up hierarchy
- Each AM a set of modules written in SPECIAL
  - Modules could be reused in more than one AM
- **Mapping specifications** define functions of one AM in terms of next higher AM
- Hierarchy consistency checker: ensured consistency among hierarchy specs, associated module specs for AMs, mapping specs between AMs
Design Hierarchy

• Look at each pair of consecutive AMs, mappings between them

• For each function in higher AM, write programs to show how it was implemented in terms of lower-level AM
  • Written in high-order language
  • Translator mapped program into common internal form that HDM tools used
  • Specs mapped into intermediate language; this and common internal form generated verification conditions
    • Sent to Boyer-Moore theorem prover
  • If lower-level AM correct, then higher-level AM verified to work correctly
Verification in HDM

• Approach: prove the FTLS correctly implemented predefined properties within a model

• Used to verify design of a multi-level security (MLS) tool implementing a version of Bell-LaPadula model (called SRI model)
SRI Model

• Some SRI model entities had no corresponding Bell-LaPadula features
  • Visible function references and results (VFUN, OVFUN)
  • Defined subjects implicitly (function callers)
  • *-property addresses downward flow of information

• Bell-LaPadula model had features SRI model did not
  • Discretionary access control, current access triples
  • Defined subjects explicitly
  • *-property addressed allowable downward access
Properties of SRI Model in MLS Tool

- Information returned by specific function invocation to subject can depend only on information with security levels no greater than subject.
- Information flowing into state variable (i.e., VFUN) can depend only on other state variables with security levels no greater than that of first state variable.
- If value of state variable modified, only function invocation with security level no greater than level of state variable can do the modification.
MLS Tool

- Processed SPECIAL specification describing external interfaces to SPECIAL model
  - One AM represented, so no mappings
  - Could be multiple modules in specification; each module had to be verified, and then the set verified using hierarchy consistency tool
MLS Tool

• To verify properties:
  • MLS tool generated formulae claiming correctness of properties
  • Property 1 correctness: formulae generated from exceptions from visible functions and VFUN, OVFUN return values
  • Properties 2, 3 correctness: formulae generated for each new value assignment to state variables

• Formulae (verification conditions) submitted to theorem prover

• Theorem prover reported the verification conditions that passes, failed, could not be proven
Boyer-Moore Theorem Prover

- User provides theorems, lemmata, axioms, assertions needed for proof
  - For example, rules of reflexivity, associativity, transitivity among partial ordering relations
  - Provided in a LISP-like notation
  - Maintained list of previously proven theorems, axioms for future proofs
- Used extended propositional calculus
- Heuristics organized to find proof in most efficient manner
  - Used a series of steps on formula in search of proof
Boyer-Moore Steps

- **Simplify**: apply axioms, lemmata, function definitions, and other techniques
- **Reformulate**: replace terms by equivalent terms easier to process
- **Substitute equalities**: replace equal expressions with appropriate substitutions to eliminate equality expressions
- **Generalize**: introduce variables for terms that are no longer used
- **Eliminate** irrelevant terms
- **Use induction** to prove theorems when needed
Boyer-Moore Evaluation

1. Iterated between simplify, reformulate steps until formula proved or disproved, or formula did not change
2. Substitute equalities, and if any changes then go back to step 1
3. Generalize, and if any changes then go back to step 1
4. Eliminate, and if any changes then go back to step 1
5. Apply induction, and if any changes then go back to step 1

If formula reduced to **TRUE** or **FALSE**, done; otherwise formula could not be proven
Enhanced HDM (EHDM)

EHDM addressed difficulties with HDM

1. SPECIAL not defined in terms Boyer-Moore theorem prover could use readily
   - Missing specific constructs that theorem prover needed
   - EHDM used new language, similar to SPECIAL but with the missing constructs, such as concepts of AXIONM, THEOREM, LEMMA

2. HDM theorem prover not interactive
   - EHDM theorem prover based on Boyer-Moore theorem prover, but was interactive
Gypsy Verification Environment

• Gypsy Verification Environment (GVE) focused on implementation proofs
  • Verification system tried to show correspondence between specifications, their implementation
  • Verification system could also prove properties of Gypsy specifications

• Set of tools including a Gypsy language parser, verification condition generator, theorem prover
Gypsy Language

• Combined specification language constructs with programming language (Pascal base)

• Limitations on Pascal base
  • Could not nest routines, but could group them together in named “scope”
  • No global variables; only constants, types, functions, procedures visible between routines
  • Parameters all constant and passed only by reference
  • No pointers
  • New data structures sets, sequences, mappings, buffers; new operations of addition, deletion, moving component
Gypsy Language Specifications

• Gypsy program made up of small, verifiable units
  • Functions, procedures, lemmata, types, constants
  • Proof of unit depended only on external specifications of referenced units

• Specification constructs
  • *Entry*: conditions assumed to be true when routine activated
  • *Exit*: conditions that must have been true if routine exited
  • *Block*: conditions that must have been true if routine blocked waiting on access to shared memory
  • *Assert*: conditions that had to be true at specific point of execution
  • *Keep*: conditions that had to remain true throughout execution of routine
Gypsy Language Specifications

• Gypsy supported execution of *lemmata* as separate units
  • Lemmata defined relation among functions, global constraints
  • *hold* specification defined constraint on values of abstract data type

• Expressive level
  • Existential quantifier *some*
  • Universal quantifier *all*
  • Mechanism to distinguish old, new values
  • *Validation directive* says when to prove condition: during verification, validated at runtime, or both
Bledsoe Theorem Prover

- Interactive natural deduction system using extended first-order logic
  - Allowed subgoaling, matching, rewriting
- Every loop had to be broken by at least one *assert* specification
- Each verification condition was theorem corresponding to single path of execution
  - Due to *asserts*, finite number of execution paths
  - Condition stated that specification at beginning of path implies specification at end of path
- Analyst could guide the prover, or it could be told to choose next step
Current Verification Systems

• Prototype Verification System (PVS)
• Symbolic Model Verifier (SMV)
• Naval Research Laboratory Protocol Analyzer (NPA)

(as of the publication date of this book)
PVS

• Builds on prior work at SRI, especially EHDM
• HDM, EHDM focused on proving programs correct and the full life cycle of software development
• PVS focuses on mechanically checked specifications, readable proofs
  • It does not provide a full software development environment
  • No notion of layers of abstraction, mapping between levels
• Components:
  • Specification language integrated with theorem prover
  • Theorem prover highly interactive (a “proof checker”)
  • Other tools like syntax and type checkers, parsers
PVS Specification Language

• Strongly typed, based on first-order logic, nonprocedural
• Supports defining theories
  • Statements called *declarations* identifying types, constants, variables, axioms, formulae
  • Theories reusable, some incorporated into PVS and are called *preludes*
  • Preludes provide definitions, theorems of set theory, functions, relations, ordering, properties of numbers
  • External libraries provide finite sets, coalgebras, real analysis, graphs, lambda calculus, temporal logics
Example PVS Specification

• Built-in theory; beginning of theory of rational numbers

rats: THEORY
BEGIN
rat: TYPE
zero: rat

nonzero : TYPE {x | x ≠ zero}
/ : [rat, nonzero -> rat]
* : [rat, rat -> rat]
x, y : VAR

left_cancellation : AXIOM zero ≠ x IMPLIES x * (y/x) = y
zero_times : AXIOM zero * x = zero
END rats
Example PVS Specification

• Types *rat, nonzero*
  • *nonzero* subtype of *rat* (as all members of *nonzero* are elements of *rat*, but not vice versa)

• Constant *zero* of type *rat*

• Multiplication, division functions take 2 arguments, return value of type *rat*
  • Note second argument of division must have type *nonzero*
Example PVS Specification

• Type checker checks types for an occurrence of “/” in left cancellation
  • It generates a *type correctness condition*
  • It adds this to the specification
  • TCCs must be proved in order to show theory type correct (hence called *obligations*)

• For example, here is added declaration:

```pvs
left_cancellation _TCC1: OBLIGATION
  (FORALL (x: rat): zero ≠ x IMPLIES x ≠ zero)
```
PVS Proof Checker

• Proceeds in 4 phase:
  1. *Exploratory phase*: developer tests specification proofs, revises high-level proof ideas as needed
  2. *Development phase*: developer constructs proof in larger steps, works on making it efficient
  3. *Presentation phase*: proof is sharpened, polished, checked
  4. *Generalization phase*: developer analyzes proof, lessons learned, for future proofs

• Uses goal-directed proof search
  • So it starts from the conclusion, infers subgoals
  • Process repeats until subgoals obvious to prove
PVS Proof Checker

• Inferencing applies inference rules
  • Starts with small set of rules
  • Applies mechanism to compose rules into proof strategies

• Types of rules and some examples:
  • *Propositional rules*: cut rule for introducing case splits, another rule for raising if conditionals to top level of formula, another for deleting formulae from goal
  • *Quantifier rules*: rules for instantiating existentially quantified variables with terms
  • *Equality rules*: replace one side of an equality premise with another

• Proof strategies: frequently used proof patterns collapsed into one step
  • Examples: propositional simplification, rewriting with a definition of lemma
Experiences with PVS

• Applied in many areas beyond computer security:
  • Used by NASA to analyze requirements for several spacecraft projects, avionics control
  • Used to verify microarchitectures, complex circuits, algorithms, protocols in hardware devices
  • Used to analyze fault-tolerant and distributed algorithms
SMV

• Based on Control Tree Logic that uses 2 letters for connectives:
  • First letter: “A” (along all paths), “E” (along at least 1 path)
  • Second letter: “F” (some future state), “G” (all future states), “U” (until), “X” (next state)
  • Examples: “AX” (along all possible paths to the next states), “EX” (along at least 1 path to the next states)

• Represent model in CTL as a digraph
  • Nodes represent states
  • Propositional atoms holding in a state represented by node annotations
  • Edges show possible state transitions
Example

• Model $M$ specifies system with states $s_0$, $s_1$, $s_2$ and propositional atoms $p_1$, $p_2$, $p_3$

• Possible state transitions:
  
  $s_0 \rightarrow s_1$, $s_0 \rightarrow s_2$, $s_1 \rightarrow s_0$, $s_1 \rightarrow s_2$, $s_2 \rightarrow s_2$

• Suppose $p_1$ true in $s_0$ and $s_1$, $p_2$ true in $s_2$, $p_3$ true in $s_0$, $s_2$
Example

• Unwind the graph to create a tree of all computational paths beginning at $s_0$
SMV Language

• Program specifies system, properties to be verified
• SMV tool returns *true* (specs hold for all initial states), or a trail of actions showing how it fails
• Module *min* identifies modules of program, forms root of model hierarchy
  • Individual module specifications describe set of variables
  • May be parameterized, contain instances of other modules; can be reused as needed
SMV Language

• VAR: defines variable, identifies type of variable
  • SMV supports boolean, scalar, fixed array, structured data types
• ASSIGN: assigns initial, next values to variables
  • Next values defined in terms of current values of variables
• DEFINE: assigns values to variables in terms of other variables, constants, logical and arithmetic operators, case and set operators
• INVAR: invariant of state transition system
• SPEC: CTL specification to be proved about module
• Other features:
  • Fairness constraints to rule our infinite executions
Example

• 2 concurrent processes share mutually exclusive resource
  • Define critical section of process’ code, and protocol for entry

• Model \( M \): processes \( p_1, p_2 \)

• States for each process:
  • \( n_i \): process not attempting entry
  • \( t_i \): process trying to enter
  • \( c_i \): process in critical section

• Allowed states: \((n_1, n_2), (n_1, t_2), (n_1, c_2), (t_1, n_2), (t_1, t_2), (t_1, c_2), (c_1, n_2), (c_1, t_2)\)

• Omit \((c_1, c_2)\) as both processes cannot be in critical section at the same time
Building the model

• \((t_1, t_2)\) occurs 2 times – one with the next state \((c_1, t_2)\) and the other with the next state \((t_1, c_2)\)
  • That is, first case is when \(p_1\) gets into the critical section, and the second when \(p_2\) gets into the critical section
Graph of the Model

\[
\begin{align*}
&(n_1, n_2) \\
&(t_1, n_2) & (n_1, t_2) & (n_1, c_2) \\
&(c_1, n_2) & (t_1, t_2) & & (t_1, c_2) \\
&(c_1, t_2) & & (t_1, t_2) & & (n_1, c_2) \\
& & & (t_1, t_2) & & \  
\end{align*}
\]
What to Show

• *Safety*: only 1 process at a time can be in the critical section
• *Liveness*: a process trying to enter the critical section will eventually do so
• *Nonblocking*: a process can always request to enter its critical section
From the Model . . .

• *Safety* requires that, for all paths, $c_1$ and $c_2$ cannot be true simultaneously; in CTL, $\text{AG}\neg(c_1 \land c_2)$.
  • State $(c_1, c_2)$ not defined in model, so trivially true

• *Liveness* requires that for all paths, if $t_i$ is true, then there is some future state on the same path in which $c_i$ is true; in CTL, $\text{AG}(t_i \rightarrow \text{AF}c_i)$
  • Inspection of graph shows this is true

• *Nonblocking* requires that, for every path, every state $n_i$ has a successor state $t_i$; that is, in CTL, $\text{AG}(n_i \rightarrow \text{EX}t_i)$
  • Inspection of graph shows this is true
Use of SVM

• Used to verify sequential circuit designs
• Used to verify IEEE Futurebus+ Logical Protocol Specification
• Also used to verify security protocols, finite state real-time systems, concurrent systems
NPA

• Verification system for cryptographic protocols
  • Written in Prolog

• Based on Dolev-Yao model of rewriting terms
  • Underlying assumption: adversary can read, modify, destroy any message, and can do any operation (encryption, decryption) that a legitimate user can do
  • Also assumes adversary does not know specific words (keys, messages)
  • Goal: learn those specific words

• Approach based on interactions among a set of state machines
  • User specifies nonsecure states and tries to prove they are unreachable
NPA Languages

- NPA Temporal Requirements Language (NPATRL) expresses generic requirements of key distribution, agreement protocols

- Common Authentication Protocol Specification Language (CAPSL)
  - High-level language for cryptographic authentication, key distribution protocols
  - Idea is to specify in this language, and then translators can translate it into languages for various protocol verification systems
  - NPA has CAPSL interface
CAPSL Language

• *Protocol specification* defines protocol
• *Types specification* describes encryption, decryption operations
• *Environment specification* provides specific details about the scenario in which the protocol is to be used to help in finding a proof
Use of NPA

- Used to test and verify many protocols
  - Internet Key Exchange protocol
  - Needham-Schroeder public key protocol
Functional Programming Languages

• These languages use mathematical expressions that are evaluated
  • Expressions only depend on inputs, so results (outputs, effects) not dependent on global variables, local state
  • Functions treated like any other value, so can be modified, used as input, output parameters

• These languages are well-defined, well-typed leading to simpler analyses than programs unimplemented using nonfunctional programming languages
Examples

• OCaml: programs verified by compiler prior to execution
  • Reduces programming errors
  • Used where speed, error-free functionality is critical

• Haskell: offers built-in memory management
  • Strongly typed
  • Programs tend to be shorter, leading to a program that is easier to verify

• Rust: combines speed of C programming language with functional programming language characteristics
  • Provides thread safety, prevents segmentation faults
  • Formally proved that unsafe implementations are safely encapsulated
Formally Verified Products

• As computing power increases and formal verification methods become more scalable, formally verifying products becomes more feasible

• Example: open-source seL4 microkernel
  • Designed using high assurance techniques
  • Formally verified against its own specification, including ability to enforce security properties

• Usually done by embedding hypotheses about program in the program
  • When one is encountered, it is checked; on failure, appropriate action taken
Example: SOAAP

• Security-Oriented Analysis of Application Programs uses annotations
  • Based on compartmentalization of execution
  • Describe what parts of program should be in sandbox, how they communicate

• Example: function to decipher file, put cleartext into second file
  • Annotated functions compiled into intermediate representation
  • All such file linked
  • SOAAP performs both static, dynamic control, data flow analysis to identify violations
  • Also warns if overhead added by checking causes program not to meet performance requirements
Example

```c
__soaap_var_read("decipher")
int   retval;

__soaap_sandbox_persistent("decipher")
void  decipher(fdes in, fdes out)
{
    char key[128] __soaap_private;
    if (getkey("Key:", key) < 0)
        retval = -1;
    while ((n = read(buf, 1023, in)) > 0)
        decrypt(buf, key);
    if (write(buf, n, out) != n)
        retval = -1;
    retval = 0;
}
```
Example

• `decipher` to be run in sandbox:

  `__soaap_sandbox_persistent("decipher")`

• `key` value should not be visible outside this function

  `__soaap_private`

• `retval` used to communicate success (0) or failure (−1), so `decipher` must be able to modify its value even though it is outside scope of sandbox

  `__soaap_var_read("decipher")`
Key Points

• Formal verification based on formal specifications
• HDM, EHDM use hierarchy of abstract machines and mappings between each layer
• Gypsy focused on proving properties of implementations
• PVS provides system to prove theorems about specifications using interactive theorem prover
• SMV is a model-checking tool
• NRL Protocol Analyzer verifies protocols, can identify potential attacks