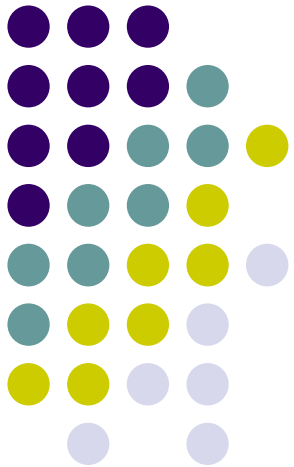
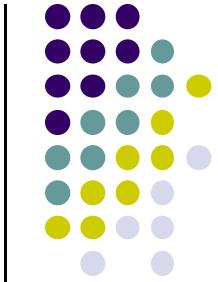
IS 2620

Formal Verification/Methods



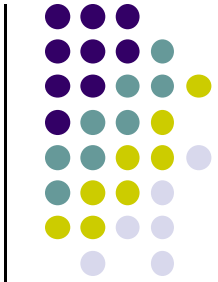
Lecture 11
Nov 5, 2014

Formal Verification



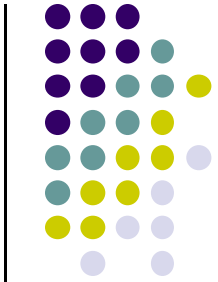
- Formal verification relies on
  - Descriptions of the properties or requirements
  - Descriptions of systems to be analyzed, and
  - Verification techniques showing requirements are met by system description
    - Rely on underlying mathematical logic system and the proof theory of that system

Formal Approach



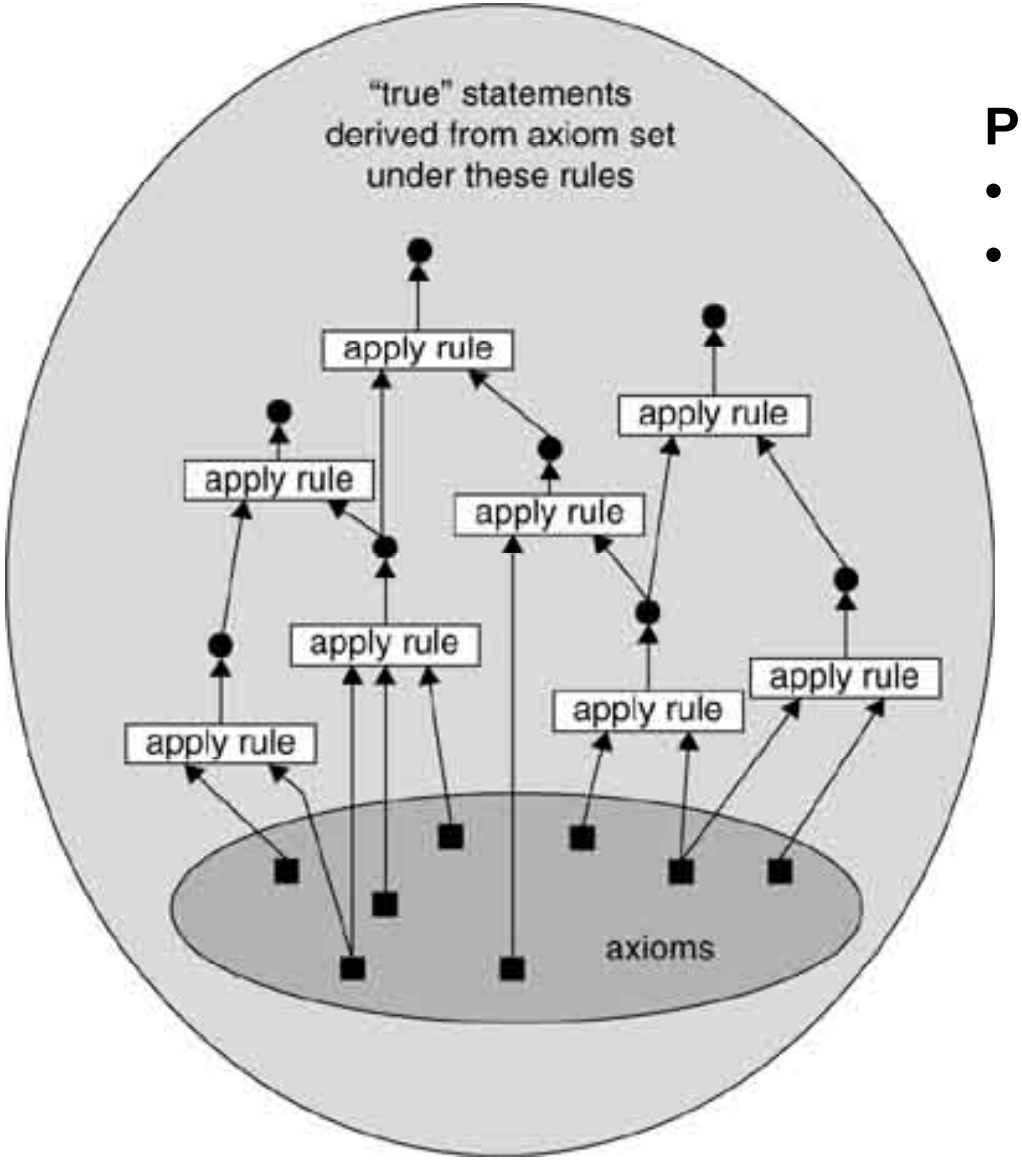
- Formal Models use language of mathematics
  - Specification languages
    - For policies, models and system descriptions
    - Well-defined syntax and semantics – based on maths
- Current trends - two general categories
  - Inductive techniques
  - Model checking techniques
    - Differences based on
      - Intended use, degree of automation, underlying logic systems, etc.

Verification techniques – Criteria for classifying



- Proof-based vs model-based
  - Proof-based
    - Formula define premises : embody the system description
    - Conclusions: what needs to be proved
    - Proof shows how to reach conclusions from premises
      - Intermediate formulas need to found to reach conclusions
  - Model-based:
    - Premises and conclusions have same truth table values
- Degree of automation
  - manual or automated (degree) & inbetween

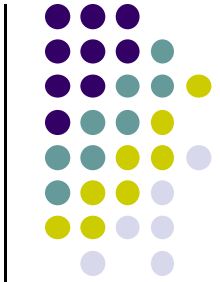
Propositional logic



Propositional
- **Axioms**
- **Inference rules**

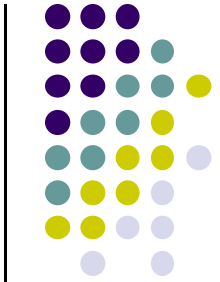
Boolean
- **And**
- **Or**
- **Not**
- **Implies**

Verification techniques – Criteria for classifying



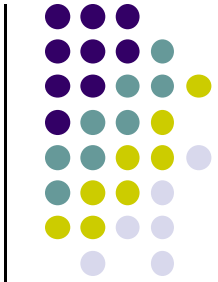
- Full verification vs property verification
  - Does methodology model full system?
  - Or just prove certain key properties?
    - Examples?
- Intended domain of application
  - HW/SW, reactive, concurrent
- Predevelopment vs post development
  - As design aid or after design

Inductive verification

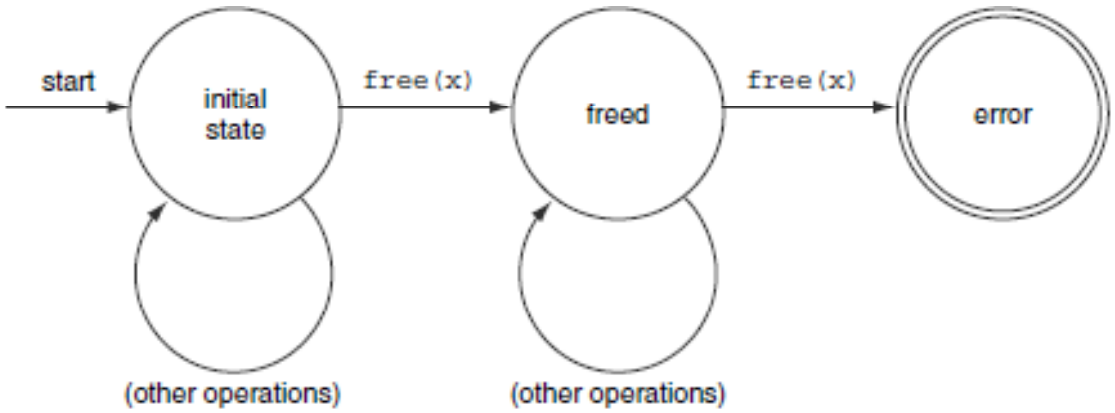


- Typically more general
- Uses theorem provers
  - E.g., uses predicate/propositional calculus
  - A sequence of proof steps starting with premises of the formula and eventually reaching a conclusion
- May be used
  - To find flaws in design
  - To verify the properties of computer programs

Model-checking

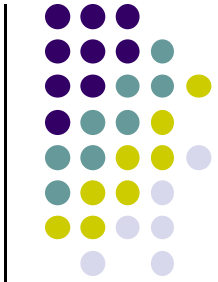


- Systems modeled as state transition systems
  - Formula may be true in some states and false in others
  - Formulas may change values as systems evolve
- Properties are formulas in logic
  - Truth values are dynamic (Temporal logic)
- Show: Model and the desired properties are semantically equivalent
  - Model and properties express the same truth table
- Often used after development is complete but before a product is released to the general market
  - Primarily for reactive, concurrent systems



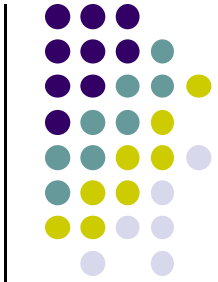
Developed primarily for concurrent/reactive systems that react to environment

Formal Verification: Components



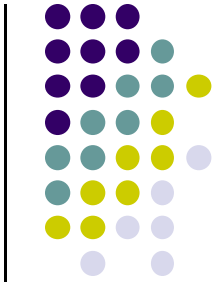
- Formal Specification
  - Defined in unambiguous (mathematical) language – precise semantics!
  - Restricted syntax, and well-defined semantics based on established mathematical concepts
    - Example: BLP Model
- Implementation Language
  - Generally somewhat constrained
- Formal Semantics relating the two
- Methodology to ensure implementation ensures specifications met

Specification Languages



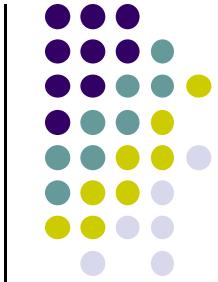
- Specify WHAT, not HOW
  - Valid states of system
  - Pre/Post-conditions of operations
- Non-Procedural
- Typical Examples:
  - Propositional / Predicate Logic
  - Temporal Logic (supports before/after conditions)
  - Set-based models
    - E.g., RBAC, formal Bell-LaPadula

Example: Primitive commands (HRU)



Create subject s	Creates new row, column in ACM; s does not exist prior to this
Create object o	Creates new column in ACM o does not exist prior to this
Enter r into $a[s, o]$	Adds r right for subject s over object o Ineffective if r is already there
Delete r from $a[s, o]$	Removes r right from subject s over object o
Destroy subject s	Deletes row, column from ACM;
Destroy object o	Deletes column from ACM

Example: Primitive commands (HRU)



Create subject s

Creates new row, column in ACM;
s does not exist prior to this

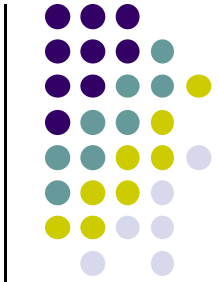
Precondition: $s \notin S$
Postconditions:

$S' = S \cup \{ s \}$, $O' = O \cup \{ s \}$

$(\forall y \in O')[a'[s, y] = \varnothing]$ (row entries for s)
$(\forall x \in S')[a'[x, s] = \varnothing]$ (column entries for s)
$(\forall x \in S)(\forall y \in O)[a'[x, y] = a[x, y]]$

Safety Theorems

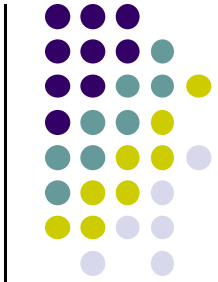
Specification Languages



- Must support machine processing
  - Strong typing
  - Model input/output/errors
- Example: SPECIAL (from SRI)
  - First order logic based; Non procedural
  - Strongly typed
  - Expressive; has capability to describe
    - Inputs, constraints, errors, outputs
    - A rich set of built-in operators

Well suite for functional specification

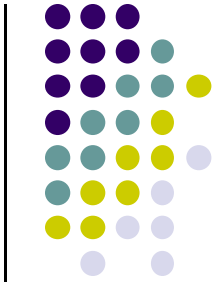
SPECIAL



- Specification modules for a system
  - Specifier defines the scope of the module
  - Provides convenience and ease of manipulation
- Sections for describing
  - Types,
    - E.g., DESIGNATOR type: Allows use of type whose specifics are to be defined at a lower level of abstraction
  - Parameters: Constants and entities
  - Assertions
    - About elements in the module
  - Functions – heart of SPECIAL
    - Statement variables and state transitions
    - Private or visible outside the module

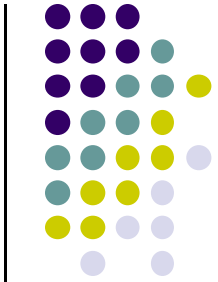
VFUN: describes variables (state)
OFUN/OVFUN: describe state transitions

Example: SPECIAL



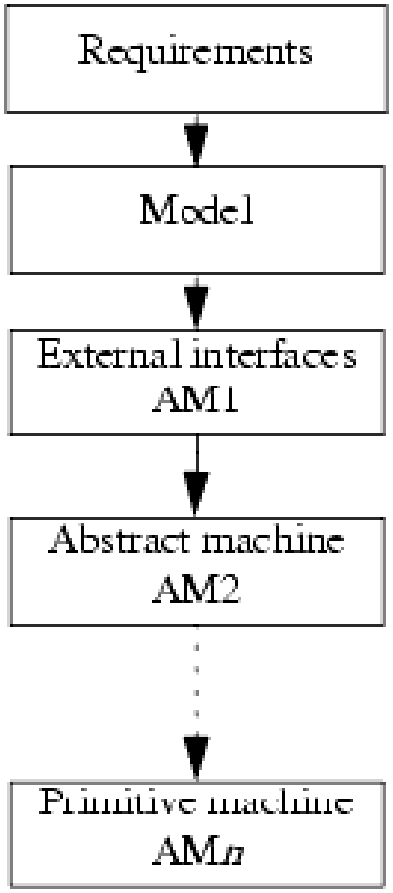
- MODULE Bell_LaPadula_Model Give_access
- Types
  - Subject_ID: DESIGNATOR;
  - Object_ID: DESIGNATOR;
  - Access_Mode: {READ, APPEND, WRITE};
  - Access: STRUCT_OF(Subject_ID subject; Object_ID object; Access_Mode mode);
- Functions
  - VFUN active (Object_ID object) -> BOOLEAN active: HIDDEN; INITIALLY TRUE;
  - VFUN access_matrix() -> Accesses accesses: HIDDEN; INITIALLY FORALL Access a: a INSET accesses => active(a.object);
  - OFUN give_access(Subject_ID giver; Access access); ASSERTIONS active(access.object) = TRUE; EFFECTS `access_matrix() = access_matrix() UNION (access);
- END_MODULE

Example: Enhanced Hierarchical Development Methodology



- Based on HDM
  - A general purpose design and implementation methodology
  - Goal was
    - To mechanize and formalize the entire development process
    - Design specification and verification + implementation specification and verification
      - Key idea; Successive refinement of specification
  - Design Spec: hierarchy of abstract machines
- Proof-based method
  - Uses Boyer-Moore Theorem Prover

Levels of Abstraction



The requirements are analyzed and accepted.

The model is proven to be internally consistent and is used as a basis for verification of the lower abstract machines.

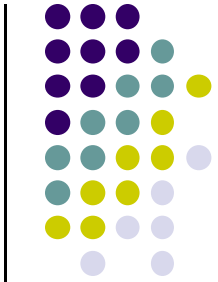
The first abstract machine is generally the external interface specification, often called a Top Level Specification (TLS) or Formal TLS (FTLS).

Formal Top Level Spec (FTLS)

Each abstract machine is mapped to successively lower-level machines, which represent successively lower levels of specification of the system.

The lowest-level specification is the so-called *primitive machine*, which is some combination of hardware and software on which the verified system runs.

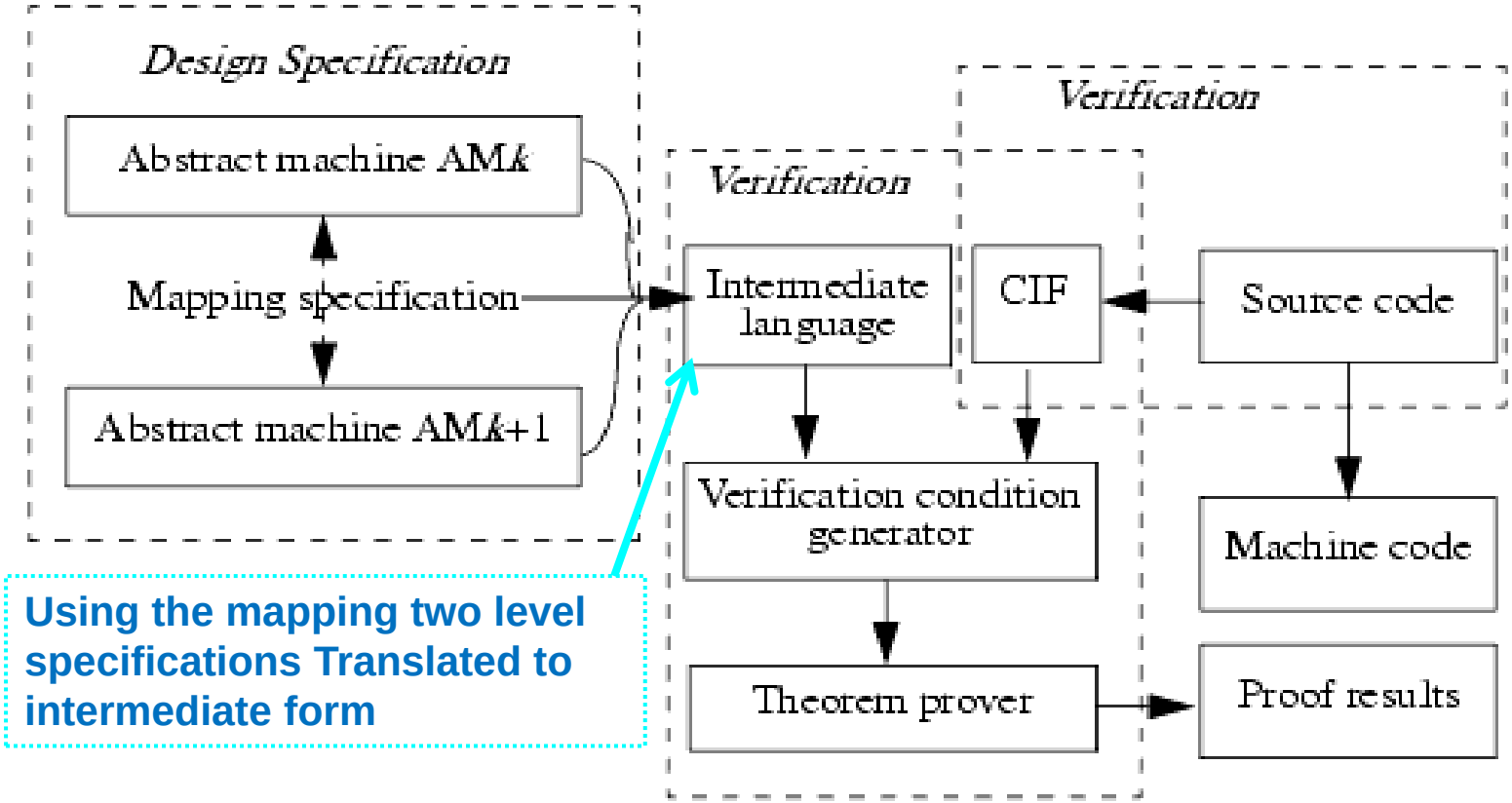
Example: Enhanced Hierarchical Development Methodology



- Hierarchical approach
  - *Abstract Machines* defined at each level
    - Hierarchy specification in in Hierarchy Specification Language (HSL)
    - AM specification written in SPECIAL
  - *Mapping Specifications* in SPECIAL
    - define functionality in terms of machines at next lower layers
  - *Hierarchy Consistency Checker*
    - validates consistency of HS, Module Spec and Mapping Spec
- Compiler : programs for each AM in terms of calls to lower level
  - that maps a program into a Common Internal Form (CIF) for HDM tools
  - Two levels of spec translated to CIF → correctness is verified (BMT prover)
- Successfully used on MLS systems
  - Few formal policy specifications outside MLS domain

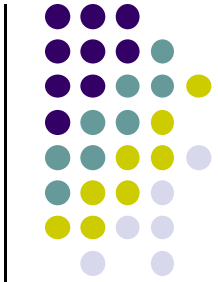
HDM Verification

Used for MLS



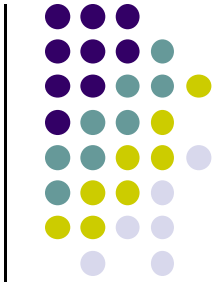
Using the mapping two level specifications Translated to intermediate form

Boyer-Moore Theorem Prover



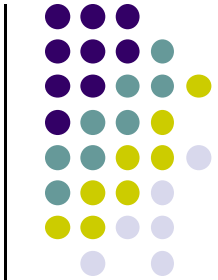
- Fully automated
  - No interface for comments or directions
  - User provides all the theorems, axioms, lemmata, assertions
    - LISP like notation
  - Very difficult for proving complex theorems
- Key idea
  - Used extended propositional calculus
  - Efficiency – to find a proof.

Boyer-Moore Theorem Prover



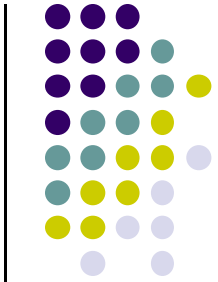
- Steps:
  - *Simplify* the formula
    - Apply axioms, lemmata, theorems
  - *Reformulate* the formula with equivalent terms
    - E.g., replace x-1, x by y and y+1
  - *Substitute* equalities
  - *Generalize* the formula by introducing variables
  - *Eliminate* irrelevant terms
  - *Induct* to prove

Gypsy verification environment (GVE)



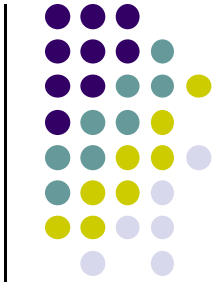
- Based on Pascal
  - Formal proof and runtime validation support
  - Focused on Implementation proofs rather than design proofs
    - verification of specification and its implementation
  - Also to support incremental development
- Specifications defined on procedures
  - Entry conditions, Exit conditions, Assertions
- Proof techniques ensure exit conditions / assertions met given entry conditions
  - Also run-time checking

Other Examples



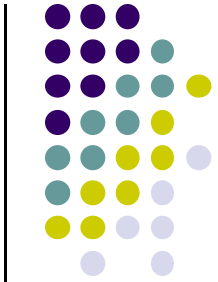
- Prototype Verification System (PVS)
  - Based on EHDM
  - Interactive theorem-prover
- Symbolic Model Verifier
  - Temporal logic based / Control Tree Logic
  - Notion of “path” – program represented as tree
  - Statements that condition must hold at *a* future state, *all* future states, all states on one path, etc.

Other Examples



- Formal verification of protocols
  - Naval Research Laboratory Protocol Analyzer
    - For Crypto protocols
      - Key management (distribution)
      - Authentication protocols
- Verification of libraries
  - Entire system not verified
  - But components known okay
- High risk subsystems

Protocol Verification



- Generating protocols that meet security specifications
  - BAN Logic
    - Believes, sees, once said
- Assumes cryptography secure
  - But cryptography is not enough