Outline

- Introduction
- Product & Version Space
- Interplay of Product and Version Space
- Intensional Versioning
- Conclusion

CM

- The discipline of managing the evolution of large and complex software systems
  - A key element in achieving process maturity
- Management support: Change control, status accounting, audit and review
- Development support: Recording configurations, maintaining consistency, building derived objects, reconstructing previously recorded configurations, constructing new configurations

Version Model

- Defines the objects to be versioned, version identification and organization, as well as operations for retrieving existing versions and constructing new versions.
  - Product Space: software objects and their relationships
  - Version Space: different versions of software objects
  - Versioned object space: combines product and version space

Product Space

- Describes the structure of a software product without taking versioning into account
- Can be represented by a product graph
  - Nodes – Software objects
  - Edges – Relationship between software objects

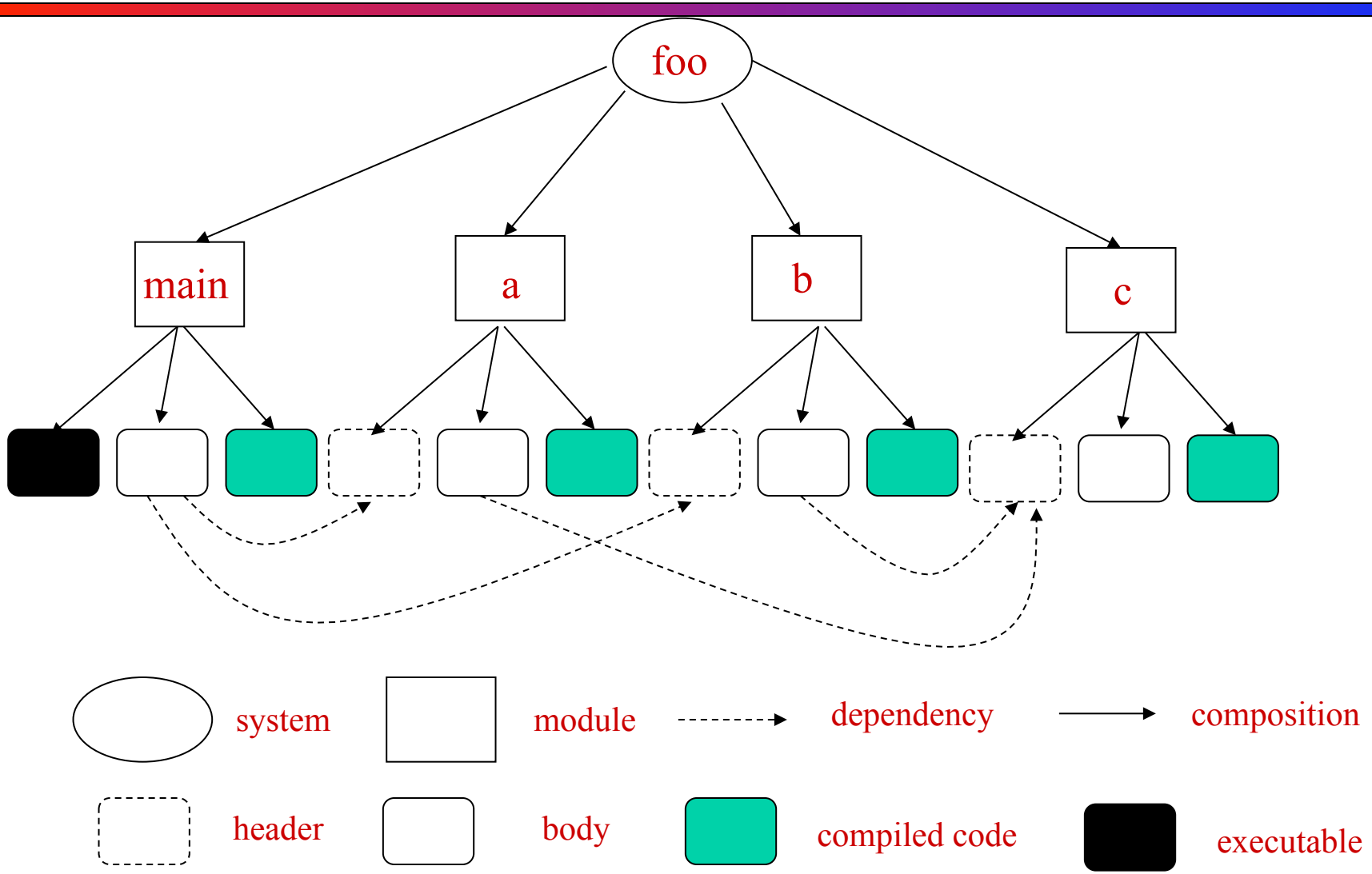
Software Objects

- Artifacts created as the result of a development or maintenance activity
  - Examples of software objects?
- Source object vs derived object
- Object identification: each software object carries a unique identifier
- May or may not have internal structure
  - For example, a program file can be stored as a text file or a syntax tree

Relationship

- Composite relationship
  - Atomic objects, composite objects, and composite hierarchy
- Dependency relationship
  - Lifecycle dependencies between requirements spec, designs, and implementations
  - Import/include dependencies between modules
  - Build dependencies between compiled code and source code

Representation



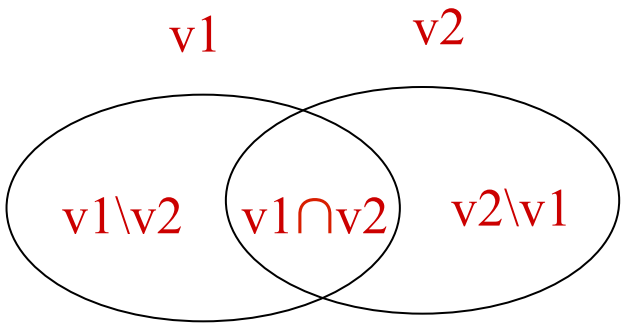
Version Space

- Represents the various versions of a single item, abstracting from the product space
  - Version: a state of an evolving item
  - Versioned item: an item that is put under version control
- Versioning can be applied at any level of granularity, ranging from a software product down to text lines

Versions

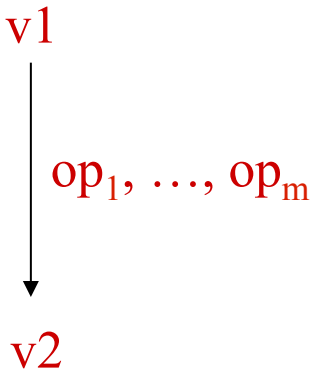
- Version identification: How to uniquely identify a version?
  - Object identifier (OID): determines whether two versions belong to the same item
  - Version identifier (VID): uniquely identifies a version within the same versioned item

Version Delta



$\Delta(v1, v2) = (v1 \setminus v2) \cup (v2 \setminus v1)$

Symmetric delta



$\Delta(v1, v2) = op_1 \ldots op_m$

Directed delta

Versioning

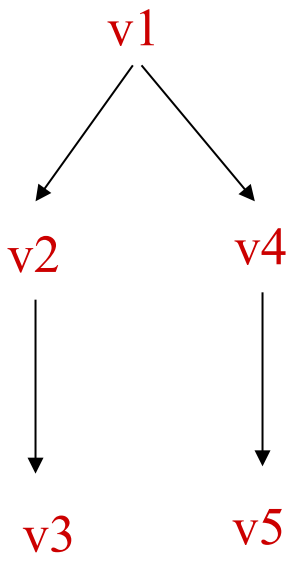
- Extensional versioning: Versions are explicitly enumerated
  - V = {v1, v2, …, vn}
  - Versions can be made immutable automatically or on demand
- Intensional versioning: Versions are implicit and constructed on demand
  - V = {v | c(v)}
  - For example, conditional compilation can construct different versions of a source file based on certain attributes

Revisions and Variants

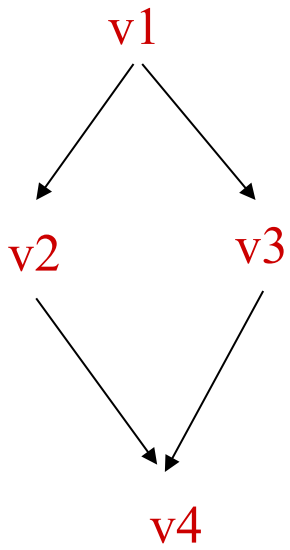
- Revision: a version intended to supersede its predecessor
  - Bug fixes, enhancements, adaptive changes
- Variant: a version intended to co-exist with its predecessor
  - For example, a software product may support multiple operating systems

Version Graph (One level)

sequence

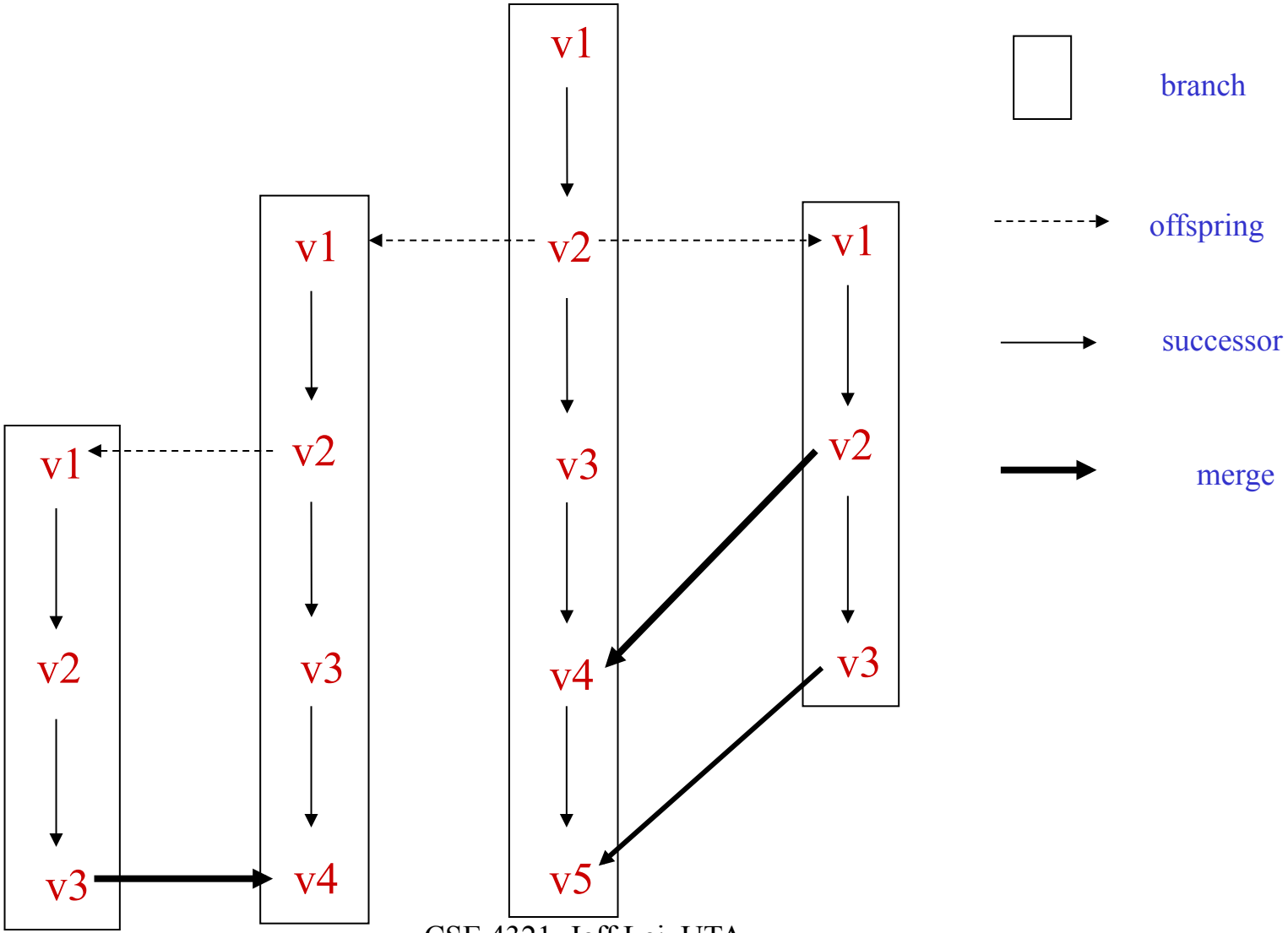


tree

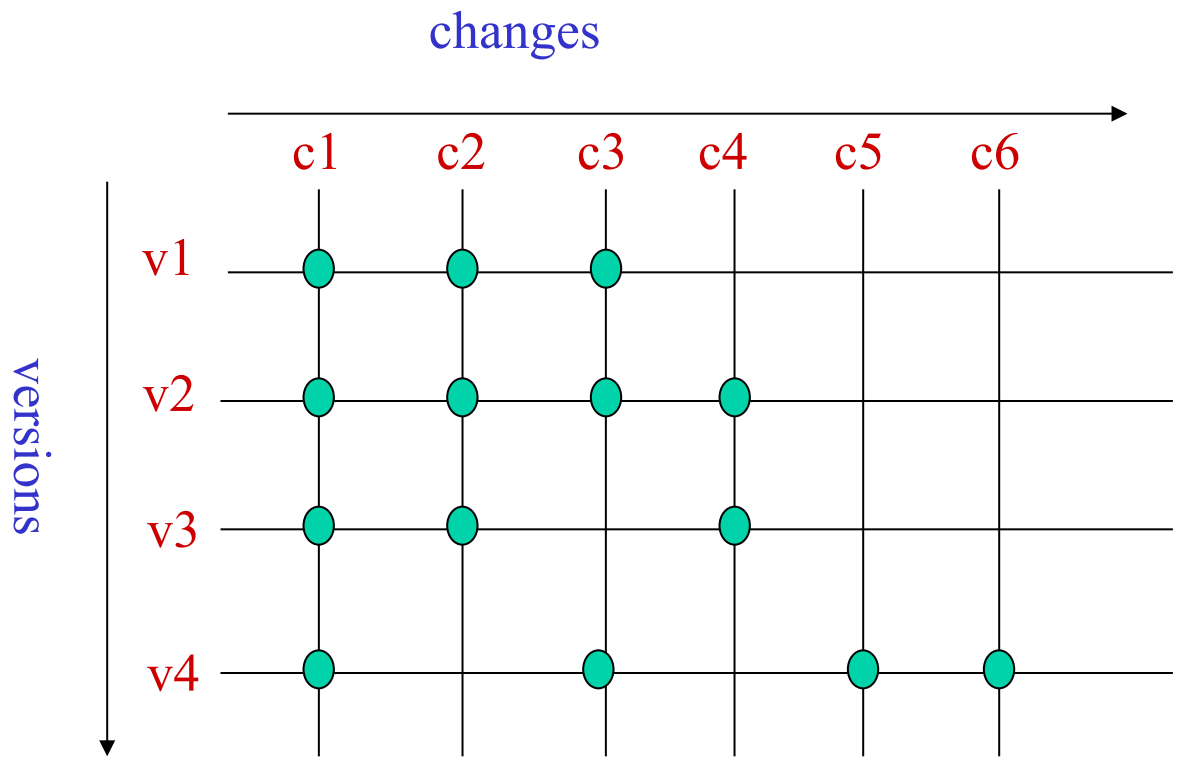


graph

Version Graph (Two levels)



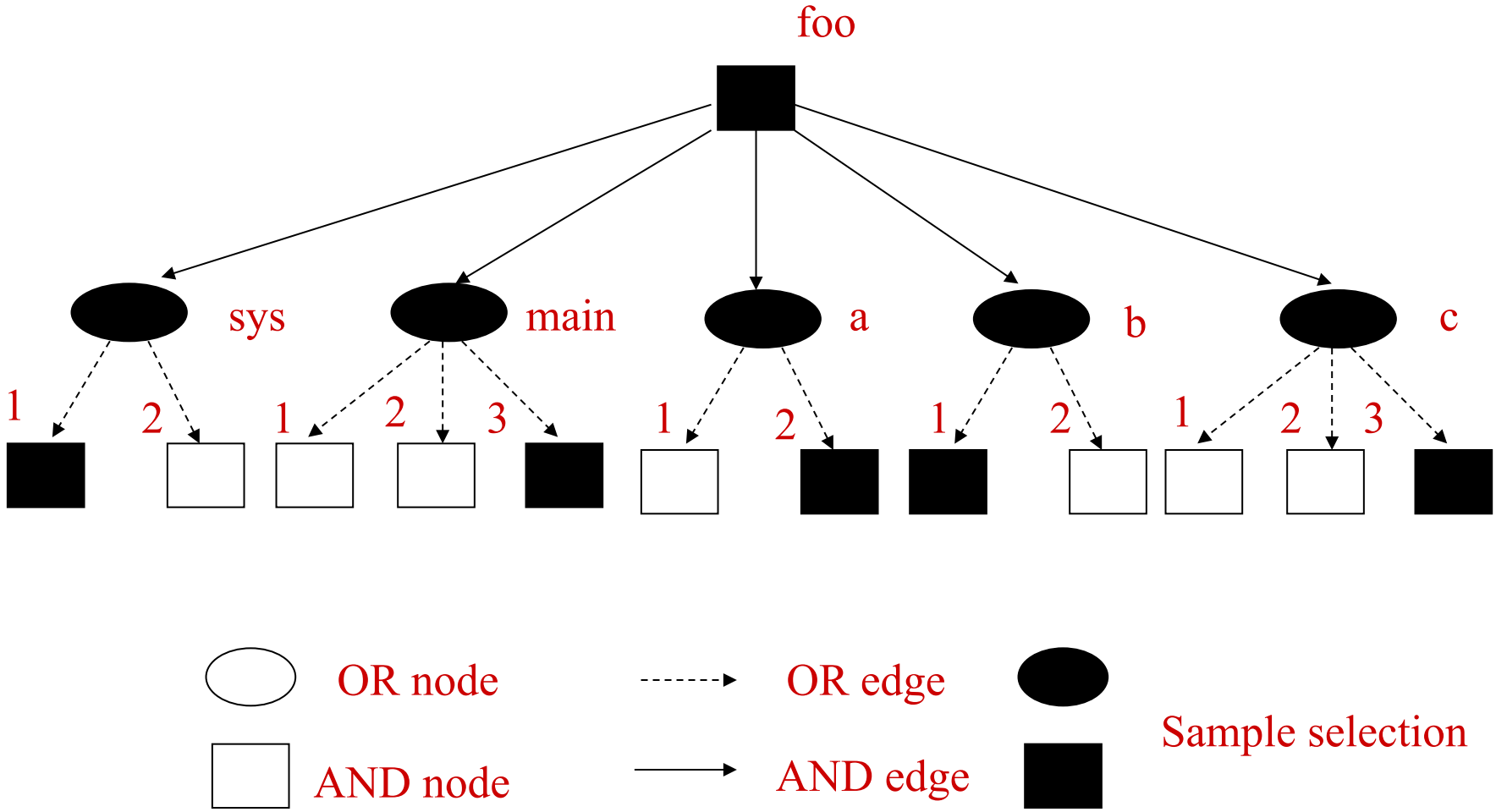
Change Space



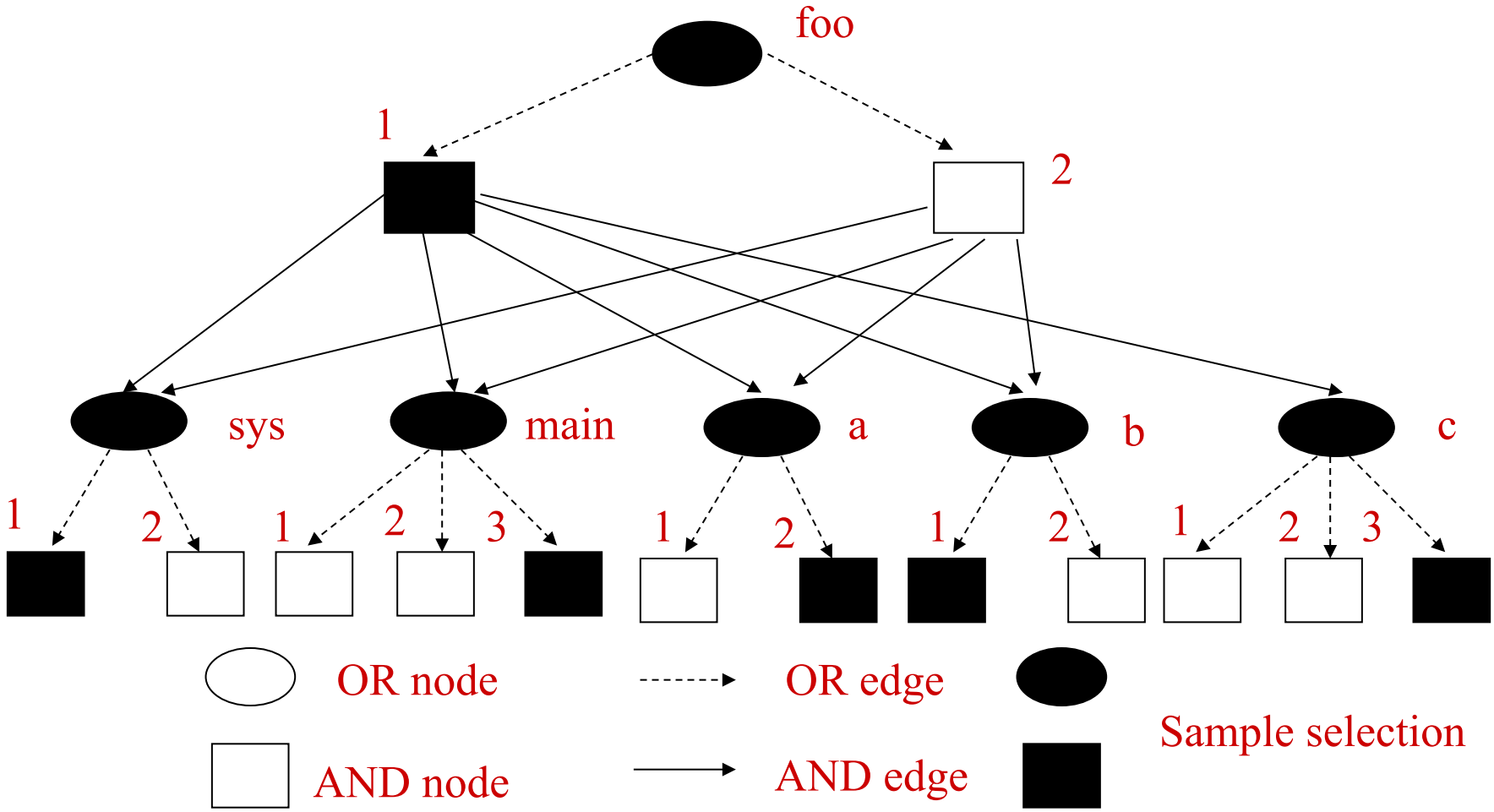
AND/OR Graph (1)

- A general model for integrating product space and version space
- AND nodes represent composition, and OR nodes represent versioning.
- Both objects and configurations can be versioned.

AND/OR Graph (2)



AND/OR Graph (3)



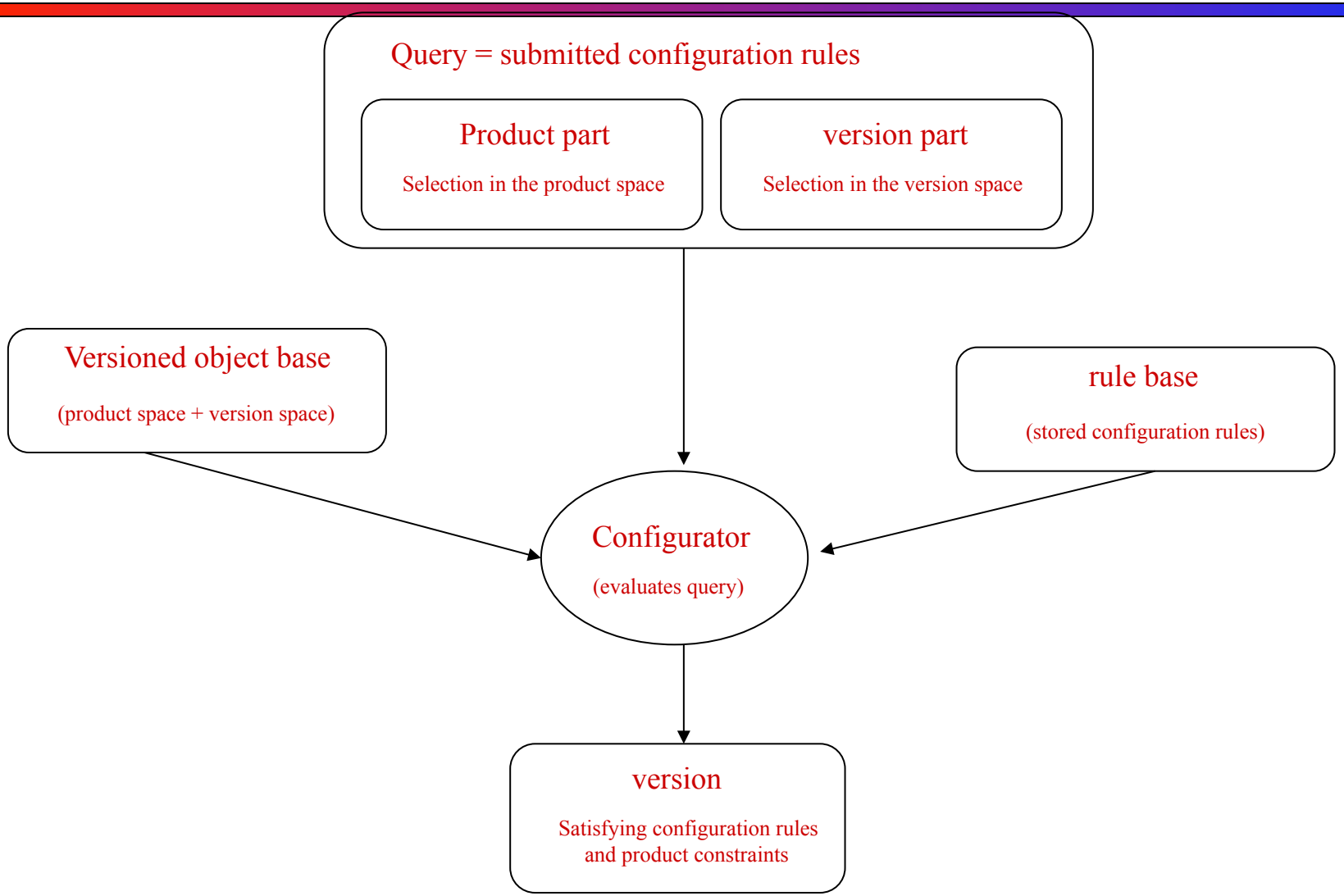
Intensional versioning

- Constructs new versions of a system from property-based descriptions
  - versions are constructed on demand by combining different changes
- When we make a product release, we need to include one version for each software object in the product
  - Assume that there are m objects and each has n version. How many possible combinations?

Consistency Control

- Configuration rules are used to rule out inconsistent combinations
- The user must be warned if a new version is created that has never been configured before
  - Quality assurance, and changes may need to make corrections

Conceptual framework



Configuration Rules (1)

- Built-in rules: hardwired into the CM system and cannot be changed by the user
  - E.g.: At most one version of a software object is contained in any constructed configuration
- User-defined rules: supplied by the user
  - E.g.: select the latest version before Nov. 10th

Configuration Rules (2)

- *revision space*

(1) t = max

(2) no = 1.1.1.1

- *variant space*

(3) os = Unix && ws = X11 && db = oracle

(4) ! (os = DOS && ws = X11)

- *change space*

(5) c1 c2 c4

(6) c2 => c1

(7) c1 ⊗ c2 ⊗ c3

Configurators

- Constructs a version by evaluating configuration rules against a versioned object base
- Rule-based: Evaluate a query against a deductive database, which consists of a versioned object base and a rule base
  - A search space of potential solutions is explored in a dept-first or breadth-first manner

Major CMs

- SCCS: Source Code Control System, Bell Labs, 1972, the first revision control system
- RCS: Revision Control System, 1980, Purdue U.
- CVS: Concurrent Versions System, 1985, client-server architecture, allows multiple developers to work together, open-source
- Subversion: Meant to be a better CVS, 2000, open source, http://subversion.tigris.org/
- ClearCase: Commercial, large-scale, distributed software development, Rational Software
- SourceForge, GitHub, BitBuckets, and others

Conclusion

- Version control is at the core of any CM, which is further at the core of any software development organization's toolset.
- The product and version space are used to represent different software objects and their different versions.
- The core issues in version control are: (1) how to represent the two spaces; (2) how to store them efficiently; (3) how to present the user a consistent view?