Reasoning about the Security of Open Architecture Software Systems

Walt Scacchi and Thomas Alspaugh
Institute for Software Research
University of California, Irvine
Irvine, CA 92697-3455 USA
14 December 2012

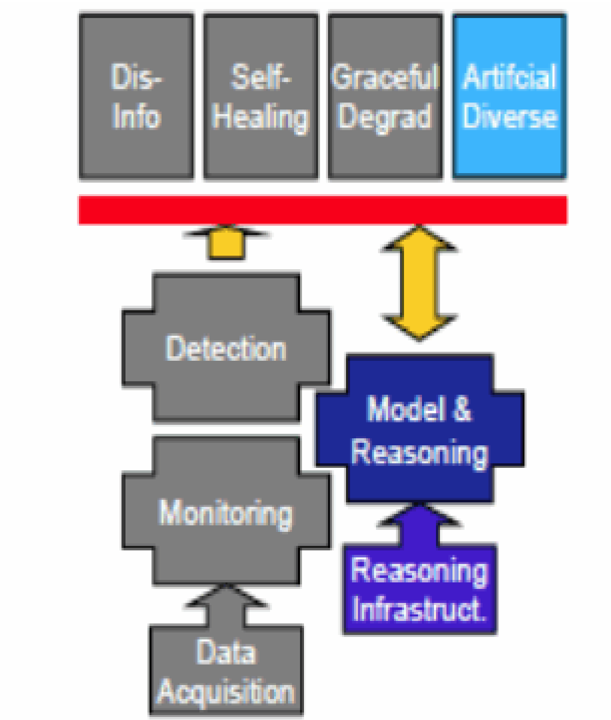
Overview

- Cyber security challenges of interest
- Reasoning about secure Open Architecture (OA) systems
- Examples of recent or work-in-progress results

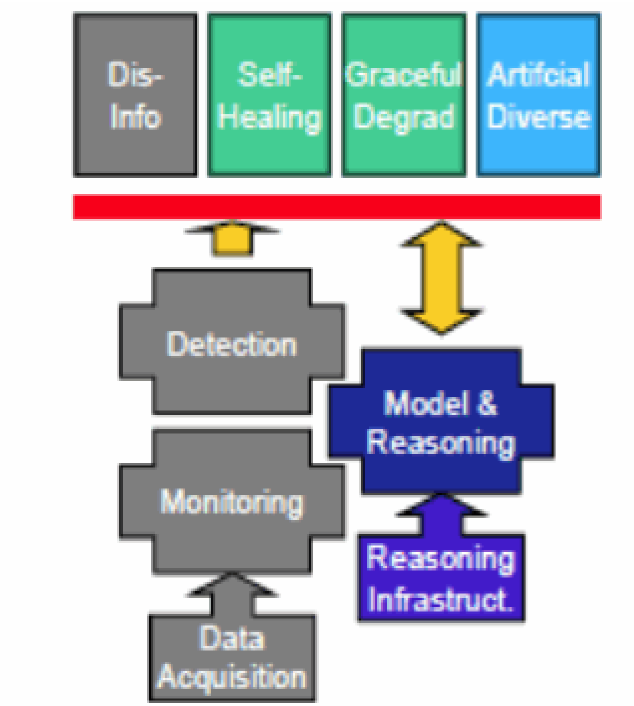
Cyber security challenges of interest

- Automated reasoning about OA system security
  - Explained in following slides
- Secure Web and Web Applications
  - Shown in demonstration case study later
- Security assuring software development environment
  - Shown in demonstration case study later

Our goal is to support reasoning about OA system security, artificial diversity, self-healing, and graceful degradation



Syst. Reasoning & Artif. Diversity



System Reasoning & Resiliency

Cyber security challenges of interest

- Automated reasoning about OA system security via robust and resilient autonomic host services
  - System self-awareness
  - Self-diagnosis (static/dynamic monitoring)
  - Self-healing (dynamic OA reconfiguration)
  - Supporting graceful degradation and artificial diversity via
    - *Functionally equivalent variants* of OA system elements
    - *Functionally similar versions* of OA system elements

System self-awareness

- Security of OA systems can be formally specified in affordable, scalable manner
  - Human readable, computer processable description of obligations and rights for access control, encryption, system updates, policies, etc.
- Requires a systematic, transparent *language* (notation) for specifying (modeling) OA system security
  - Further requires meta-model based ontology for OA system security constructs, vocabulary, logic, processing scripts, etc.
- Language model and meta-model represent relationships among elements stored in a knowledge-base (repository)
  - Relations define schemata, object attribute values stored in repository.
- We have prior research in object/knowledge-based modeling, analyzing, and simulating of self-aware software system processes

System self-awareness--*language*

- Language describes *meta-data* about elements within the system's architecture (structure) and security constraints (annotations):
  - OA system elements:
    - Components (open source software or executable binaries)
    - Connectors (data protocols, databases, middleware)
    - (Sub)System configurations (software only, s/w+h/w+net platform)
    - Ecosystem (developers, users, integrators, attackers)
  - Who, what, where, when, why, how, and with what meta-data attributes and values for each system element.

System self-awareness--*language*

- Language describes *meta-data* about elements within the system's architecture annotated with security constraints (structural model):
  - Who (actor), what (action), where (location), when (time), why (belief/policy), how (process), with what (mechanism), to what (element)
- Reasoning entails multiple modes of model analysis
  - *Static analysis*: consistency, completeness, traceability, correctness of model
  - *Dynamic analysis*: mining/filtering of computational results from monitored system (interface) execution model/states
  - *Evolutionary analysis*: detection of intended (planned) and unintended (suspicious) changes to system models, elements, or states
  - *Query processing*: retrieval of results or deductive inference from rules constraining updates to model elements, or to system execution states of interest

Sample of our prior efforts in developing languages for reasoning about features of software system architectures

Choi, S. and Scacchi, W. (2003). Formal Analysis of the Structural Correctness of Software Life Cycle Descriptions, *Intern. J. Computers & Applications*, 25(2), 91-97.

Narayanaswamy, K. and Scacchi, W. (1987). A Database Foundation for Supporting the Evolution of Large Software Systems, J. Systems and Software, 7(1), 37-49.

Alspaugh, T.A., Scacchi, W., and Asuncion, H.A. (2010). Software Licenses in Context: The Challenge of Heterogeneously Licensed Systems, *J. Association for Information Systems*, 11(11), 730-755.

Scacchi, W. and Alspaugh, T.A. (2012). Understanding the Role of Licenses and Evolution in Open Architecture Software Ecosystems, *Journal of Systems and Software*, 85(7), 1479-1494.

Scacchi, W. and Alspaugh, T.A. (2013). Advances in the Acquisition of Secure Systems Based on Open Architectures, *Cyber Security and Information Systems J.* 1(2), (to appear).

Also see our References section.

Self-diagnosis (static/dynamic monitoring)

- Static/dynamic analysis of OA system security obligations and rights in a new architectural language.
  - Annotated system has a “security architecture”
  - This architecture serves as a (design-time) *reference model*
  - Reference model guides dynamic run-time analysis of system
  - Run-time analysis determines matches, mis-matches, and conflicts with security constraints in reference model
  - Diagnostic classification rules and triggered processing scripts specify actions to take (logging, reporting, hot swap system run-time configuration, shift execution to different processor cores, etc.)

Self-healing (dynamic OA reconfiguration)

- It is now possible to develop a new language to specify, design, build, package, and deploy concurrent, multi-version systems using:
  - Multiple *functionally equivalent* executable <u>variants</u>
    - new patented compilation technology developed at UCI
  - Multiple *functionally similar* component <u>versions</u>
    - Alternative releases of software component
    - Alternative producers of competing components
      - Ex. Web browsers: Internet Explorer, Google Chrome, Apple Safari, and Mozilla Firefox
      - Ex. Office apps: MS Word, AbiWord, Google Docs, OpenOffice Write
  - Concurrent software build processes enable creation of OA system configured with different variants or versions for one or more processors (or processor cores)

• Self-healing (dynamic OA reconfiguration)

- Availability of multiple concurrent system variants or versions can enable hot/cold swapping of executable run-time systems
  - Requires automated processing scripts and configuration rules
  - Such scripts and rules need to be part of the OA system modeling language
  - Multiple alternative configurations can be stored in repositories in ready-to-swap state, or ready to be adapted within remote target system.

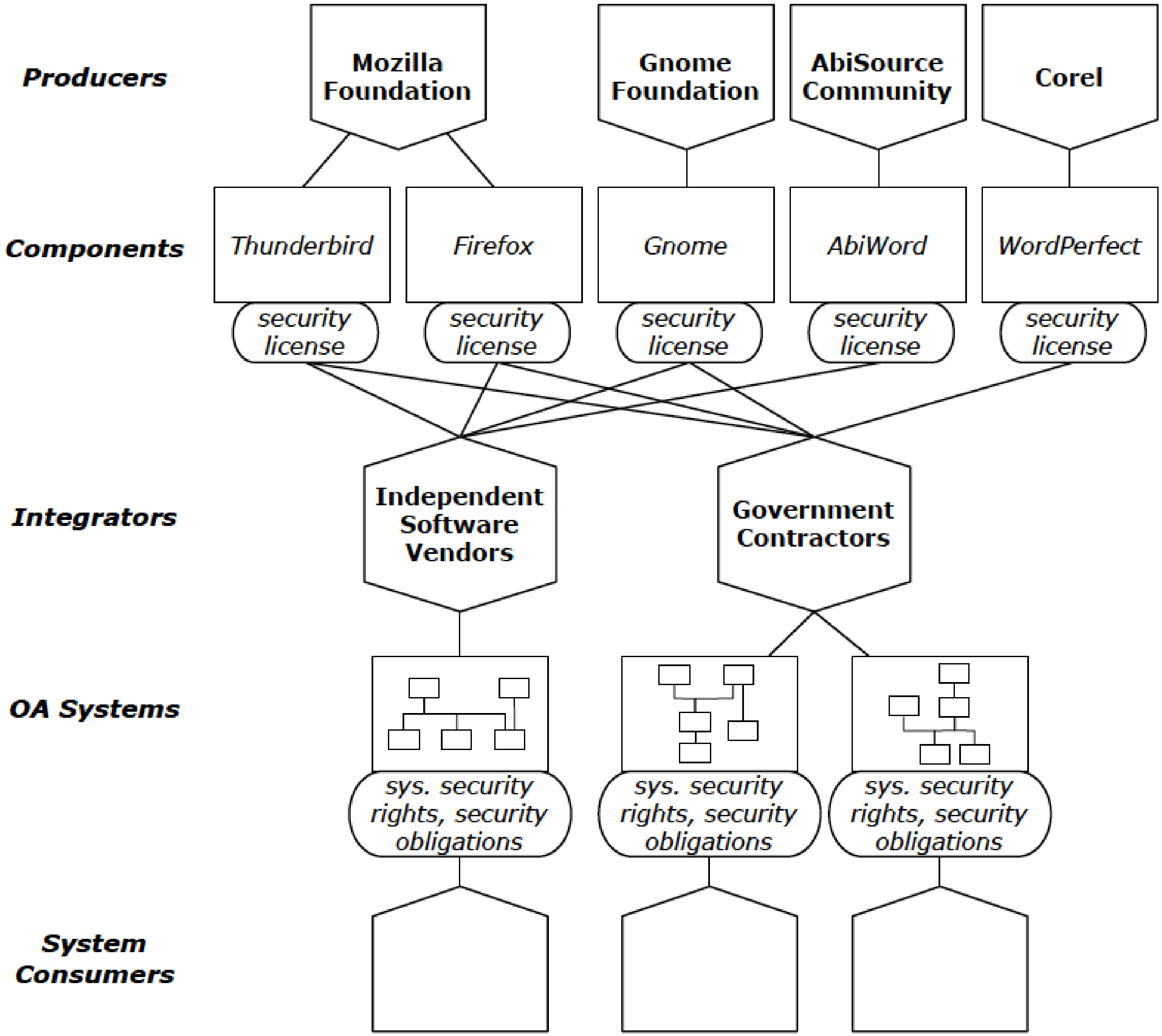
Supporting graceful degradation and artificial diversity

- Hot (dynamic) or cold (static) swappable OA elements
- *Functionally equivalent variants* of OA system elements
  - e.g., executable components with different memory maps, created via multiple concurrent compilations
- *Functionally similar versions* of OA system elements
  - e.g, software product line component instances from different producers
  - e.g., multiple concurrent system configurations with different interconnection layouts

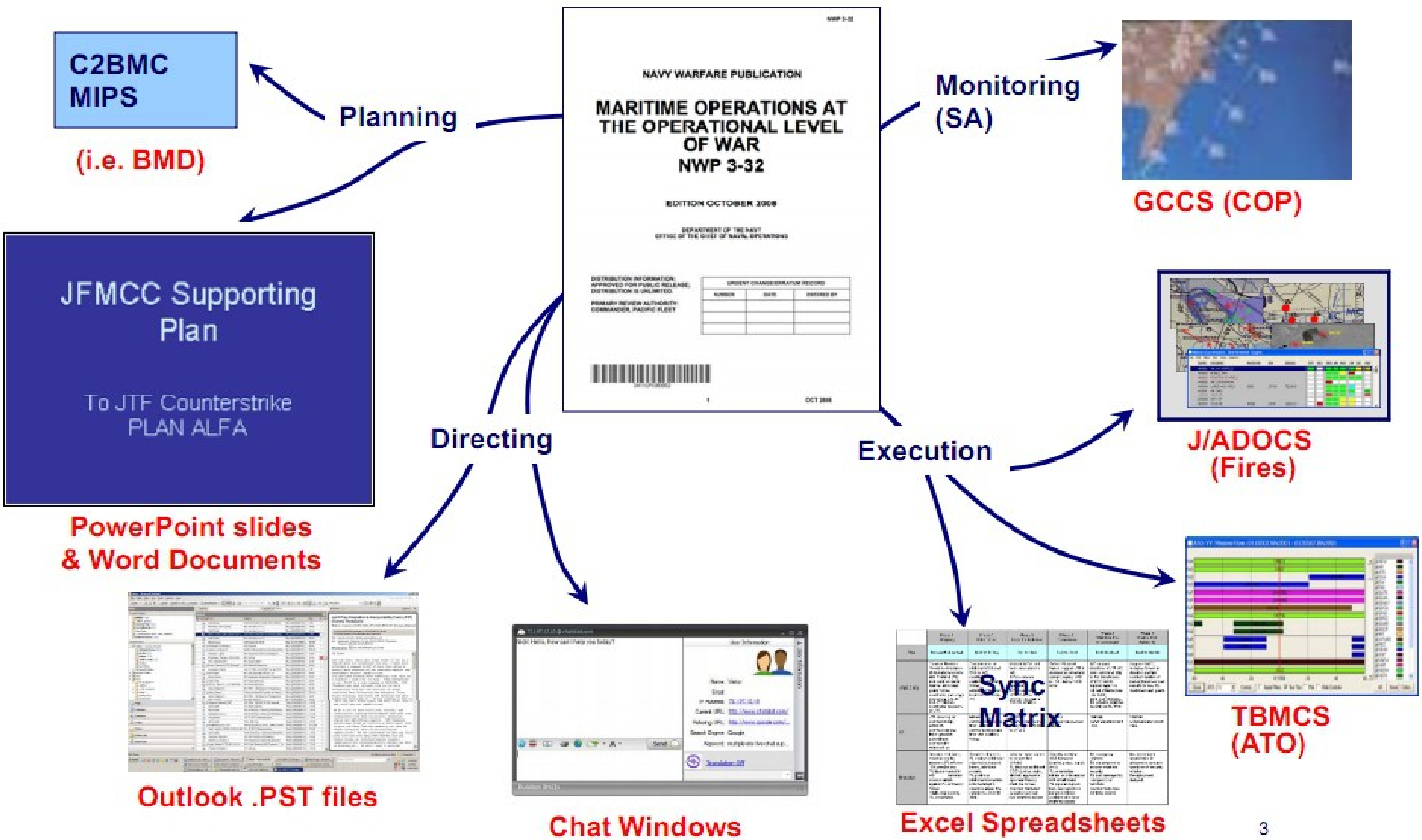
Examples of recent or work-in-progress results

- Language for modeling and meta-modeling as basis for reasoning about OA obligations and rights (simple constraints—not security)
- Tools and techniques for automated reasoning about OA obligations and rights integrity
- OA system software development environment
  - Based on *Eclipse* with *UCI ArchStudio* and analysis plug-in modules

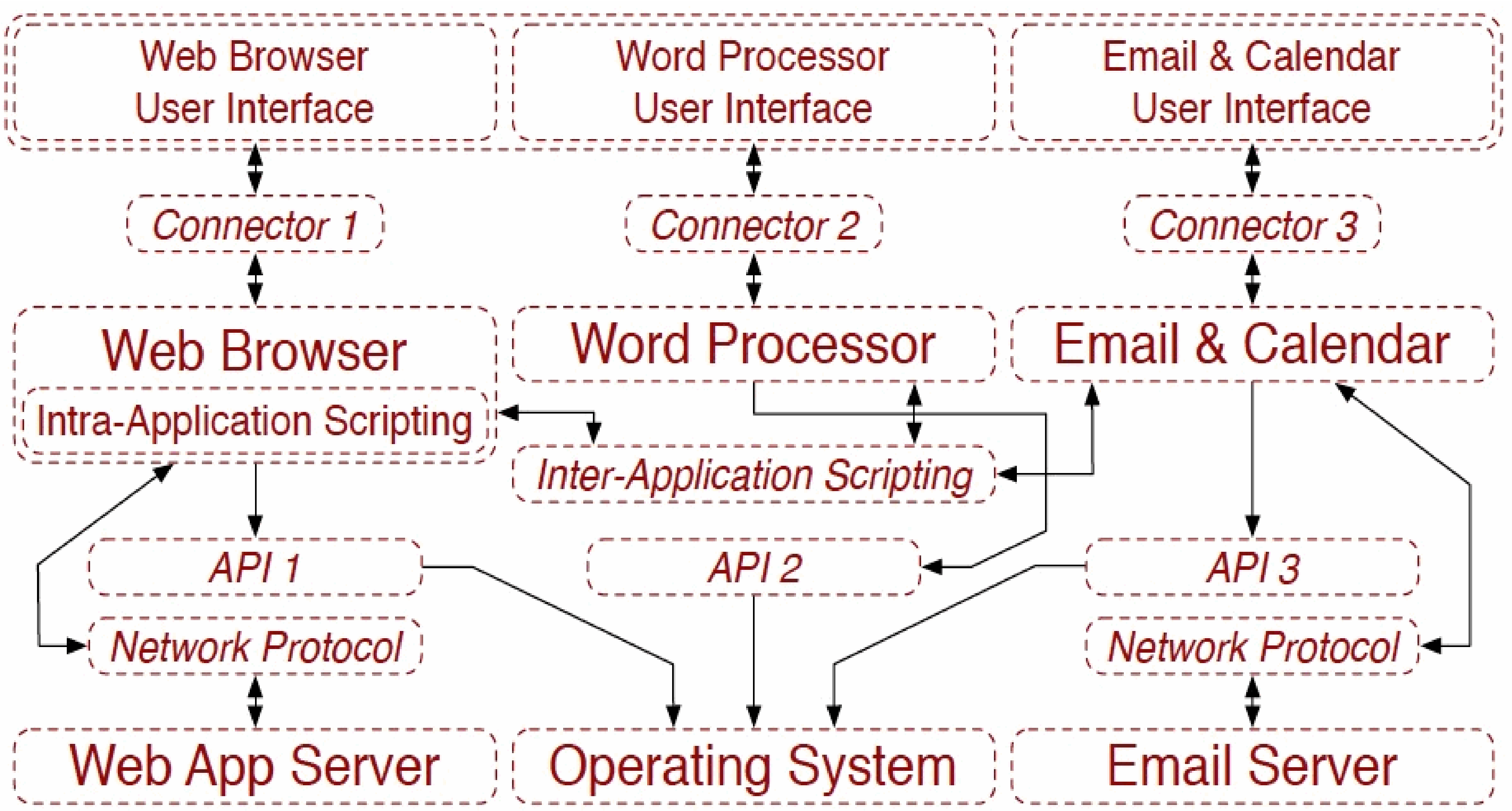
Software ecosystem of OA system producers, integrators, consumers



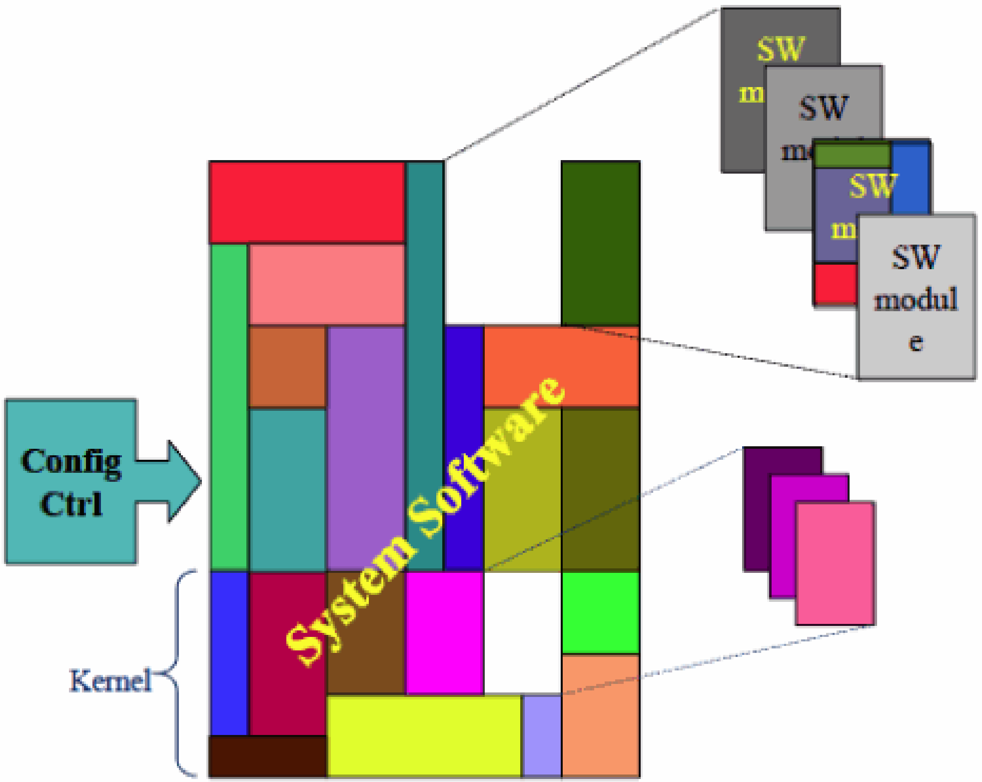
Conceptual architecture for Web-based command and control system (C2RPC)



Design-time view of a C2RPC-like OA system



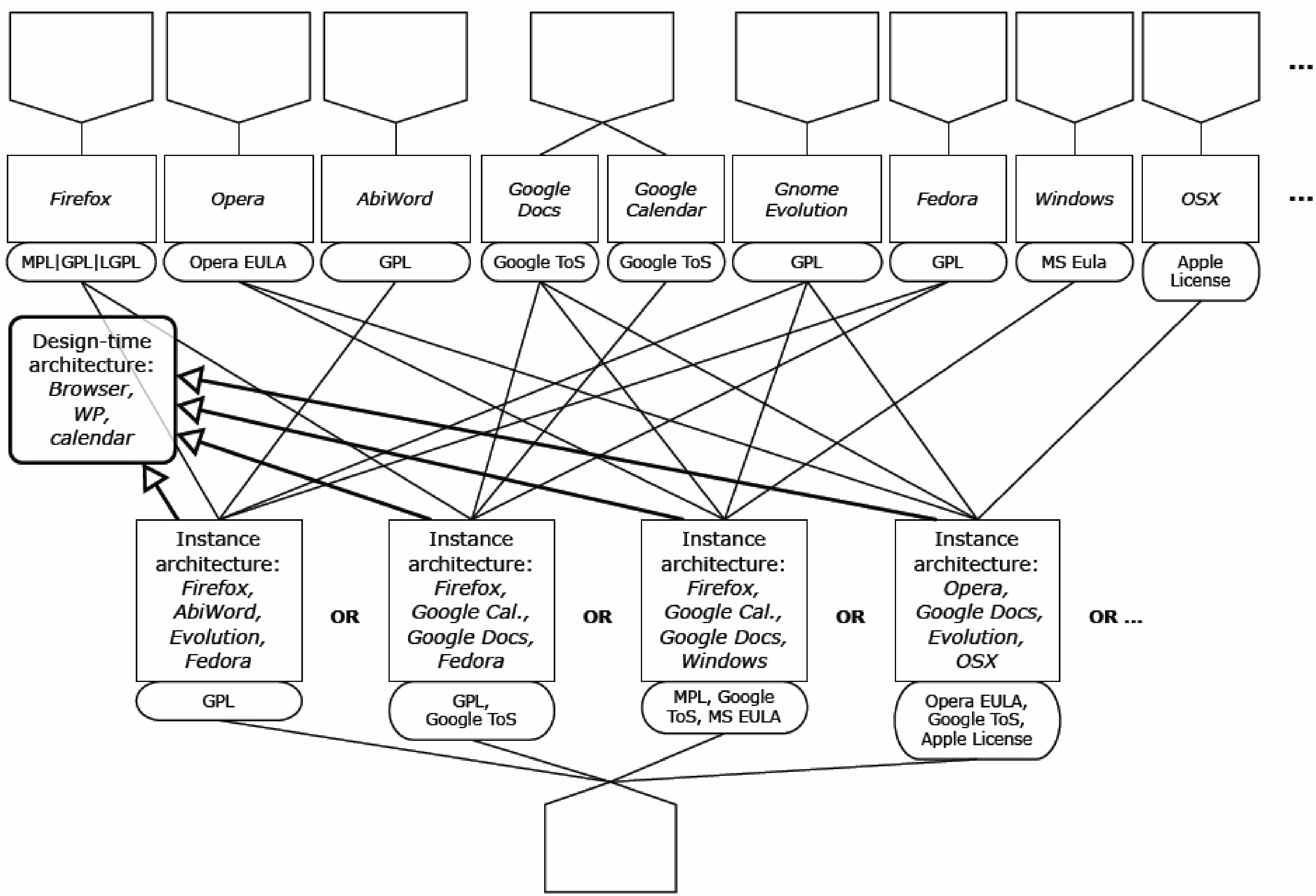
Software product line of *functionally equivalent* variants or *functionally similar* OA system versions enables support for artificial diversity and dynamic reconfiguration



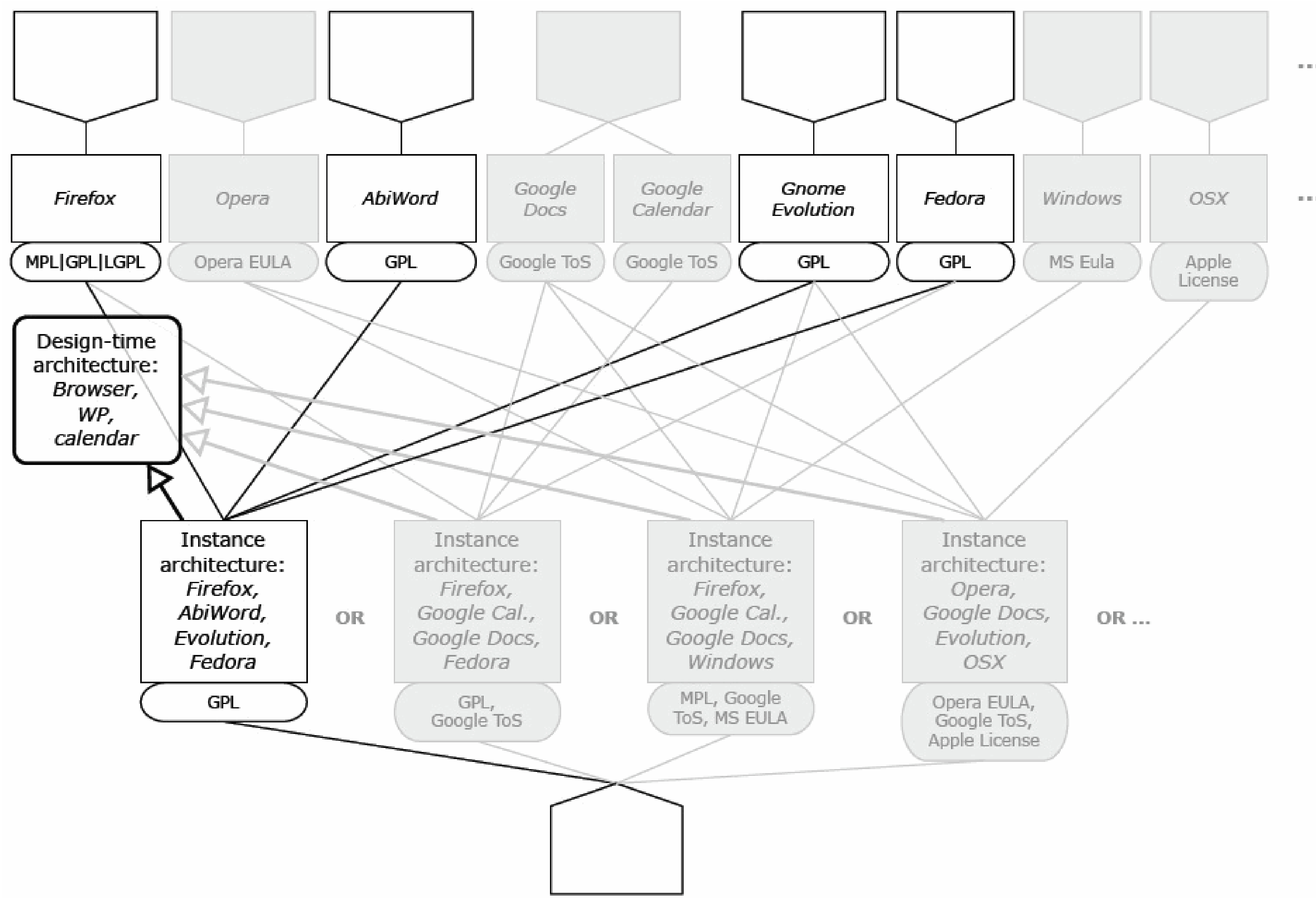
Artificial Diversity within system software (dynamic reconfig)

- Many implementations of a module
- Alleviate vulnerability due to implementation
- Diverse configuration reduce compromise propagation
- Re-configuration can mitigate compromise
- May or may not need event detection

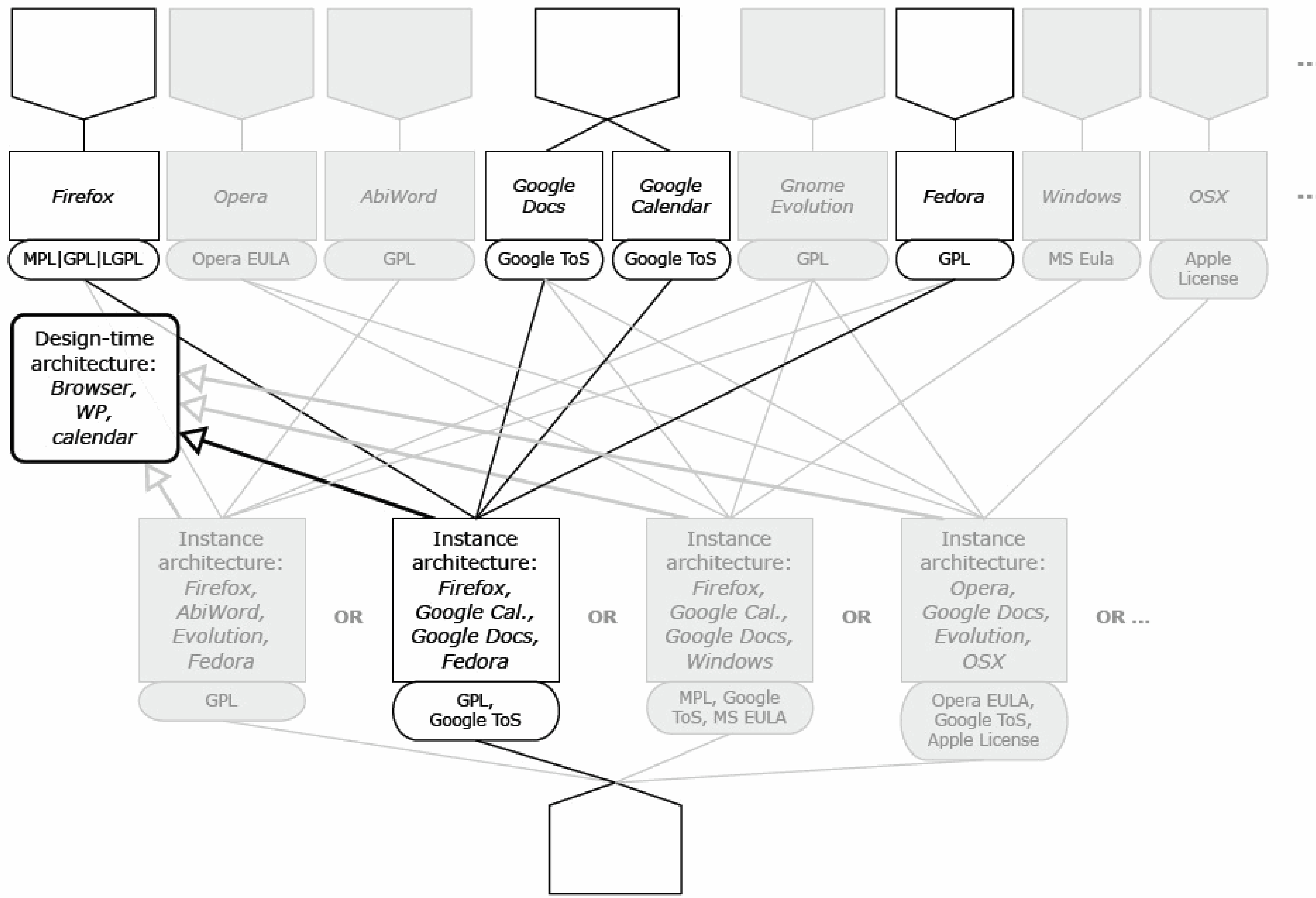
Software product line of *functionally similar* OA system alternatives



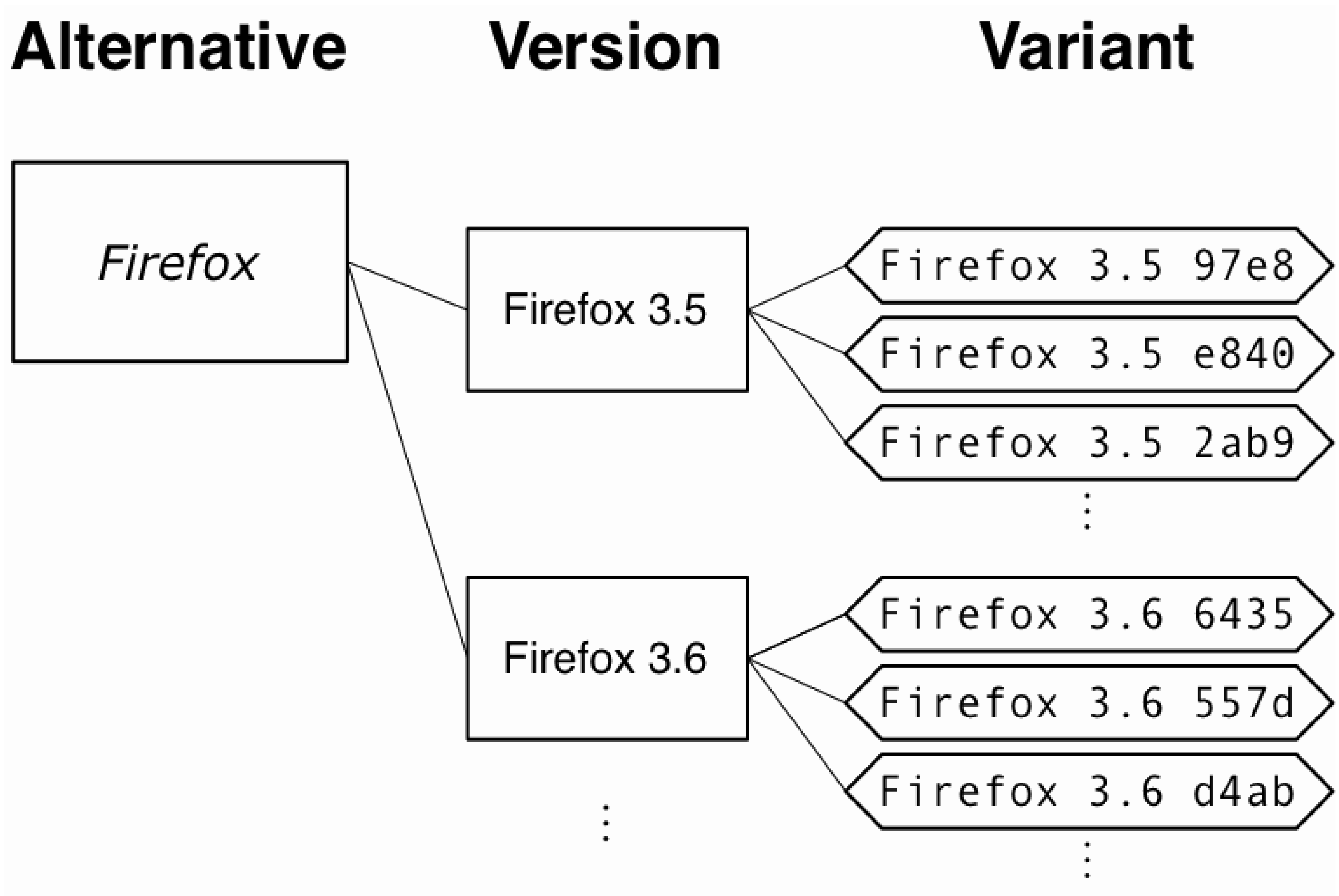
Product line selection of one alternative system configuration



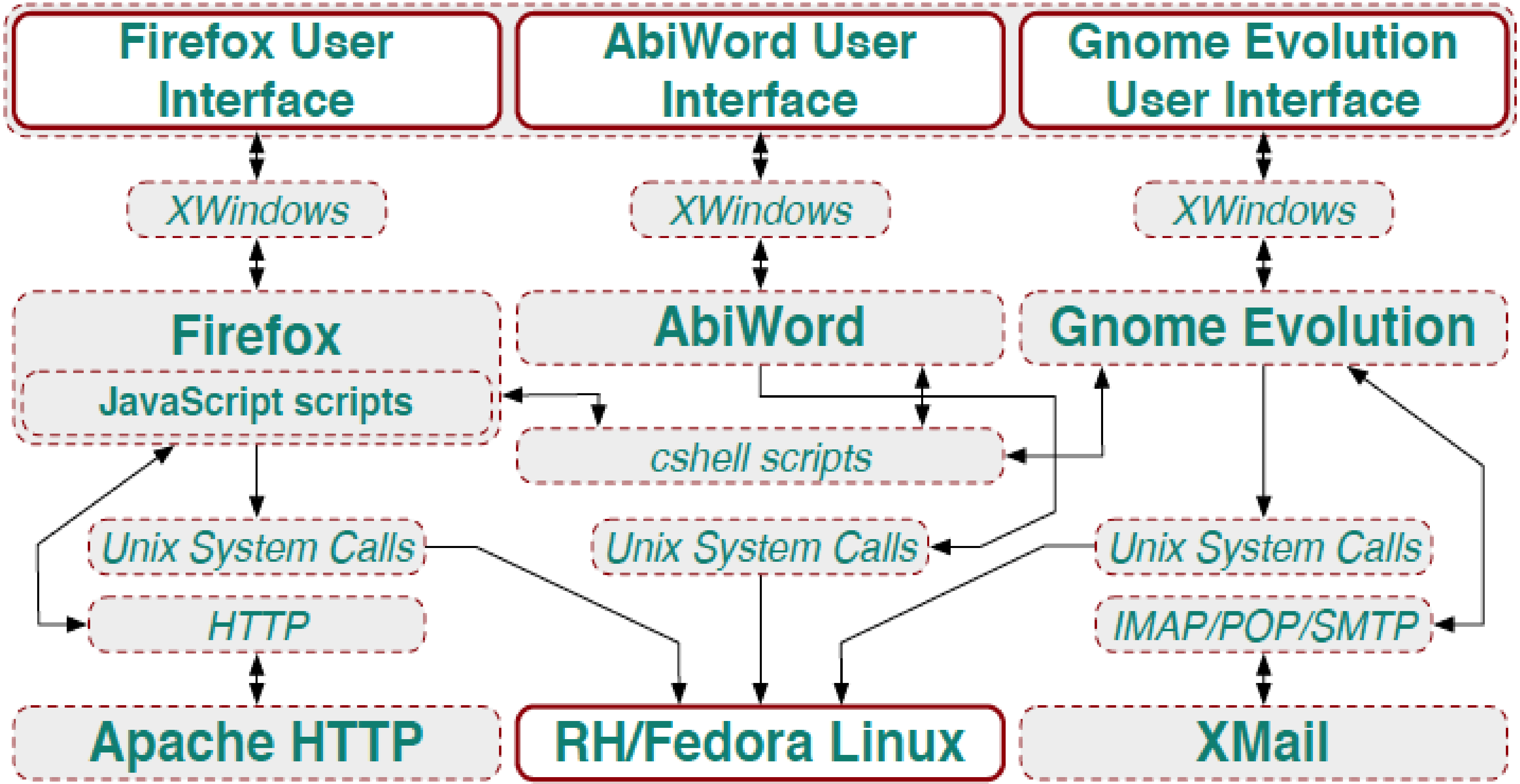
Product line selection of different functionally similar alternative



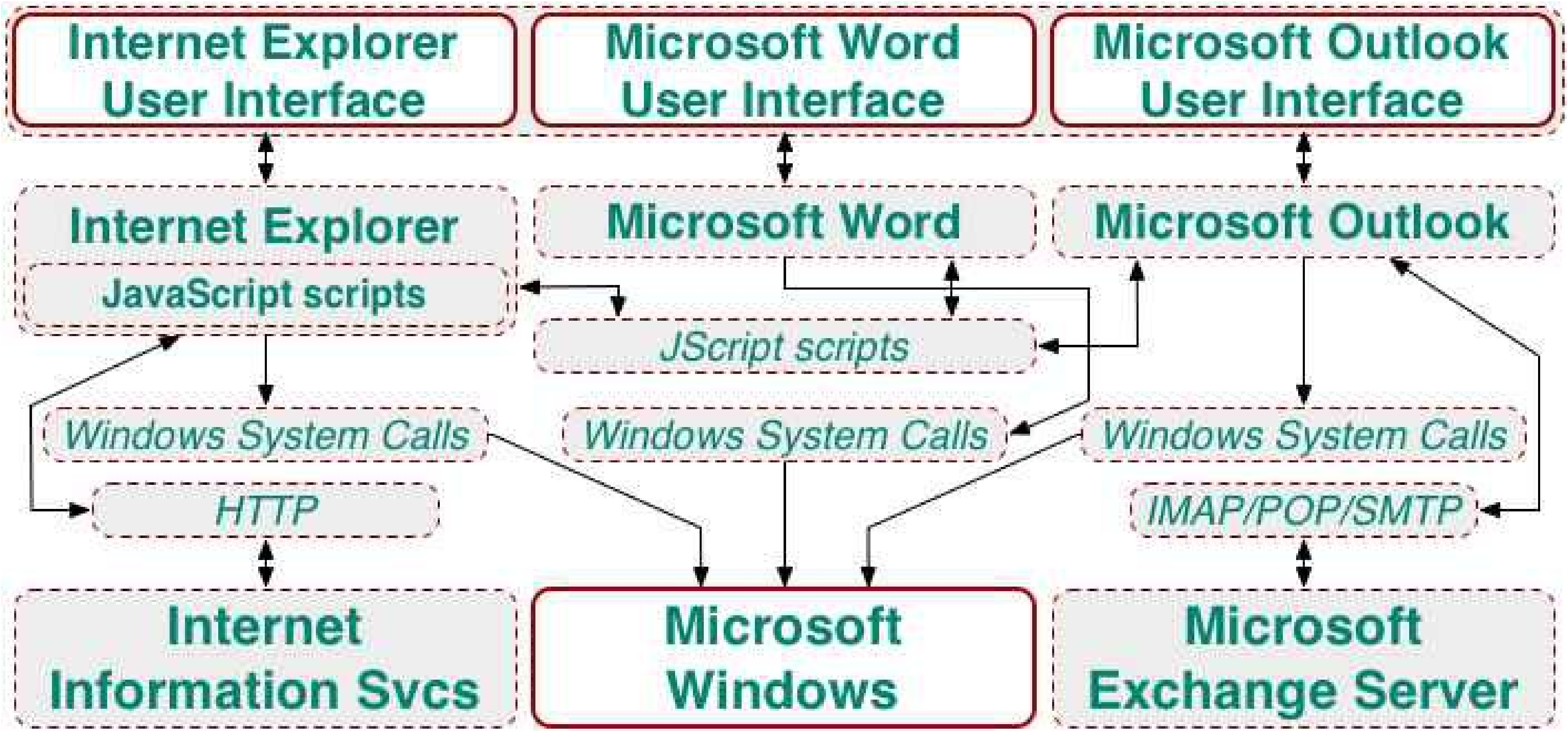
One Web browser component alternative, versions, and instance variants for inclusion via dynamic reconfiguration of OA system



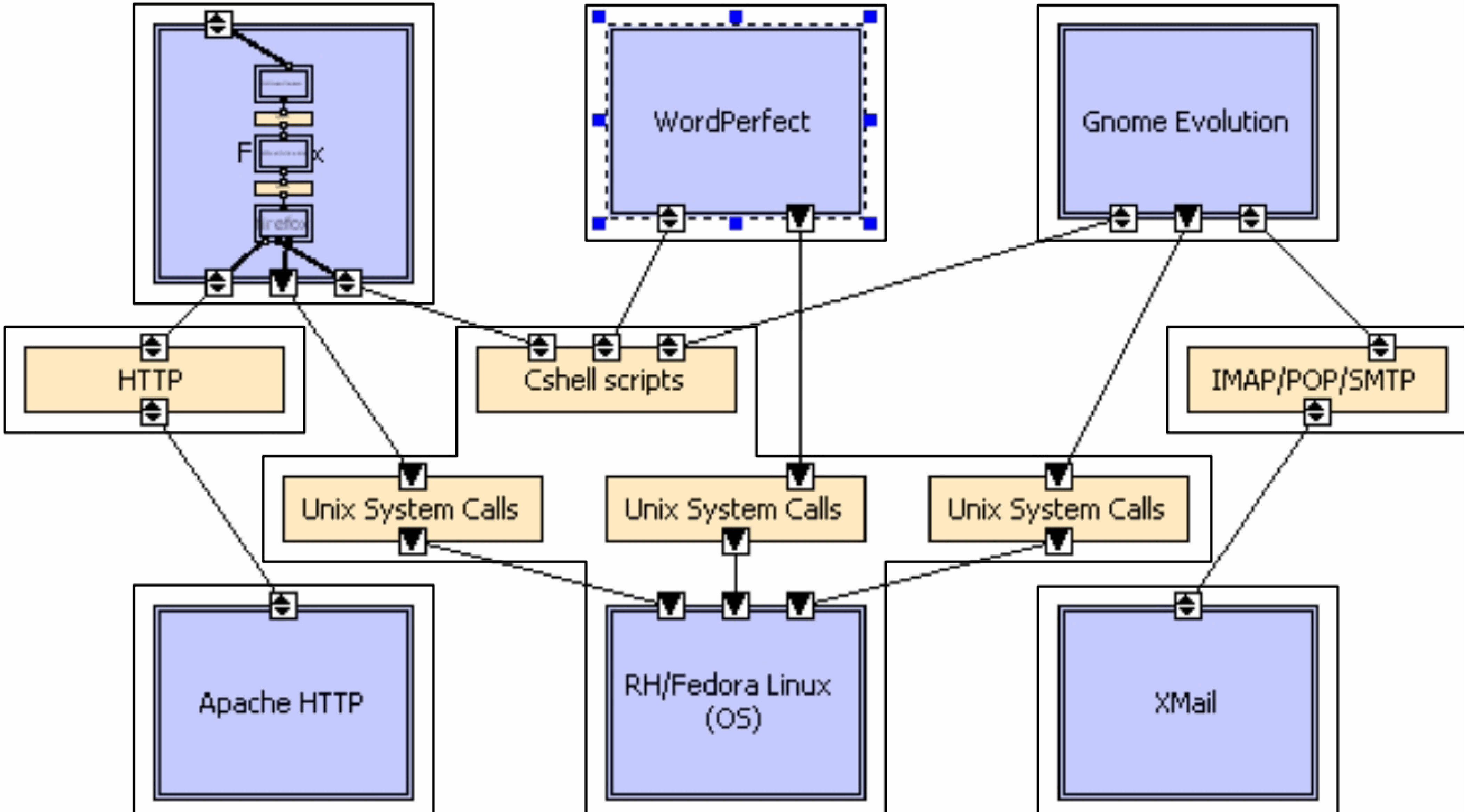
Build-time view of OA design selecting *OSS* product family alternatives



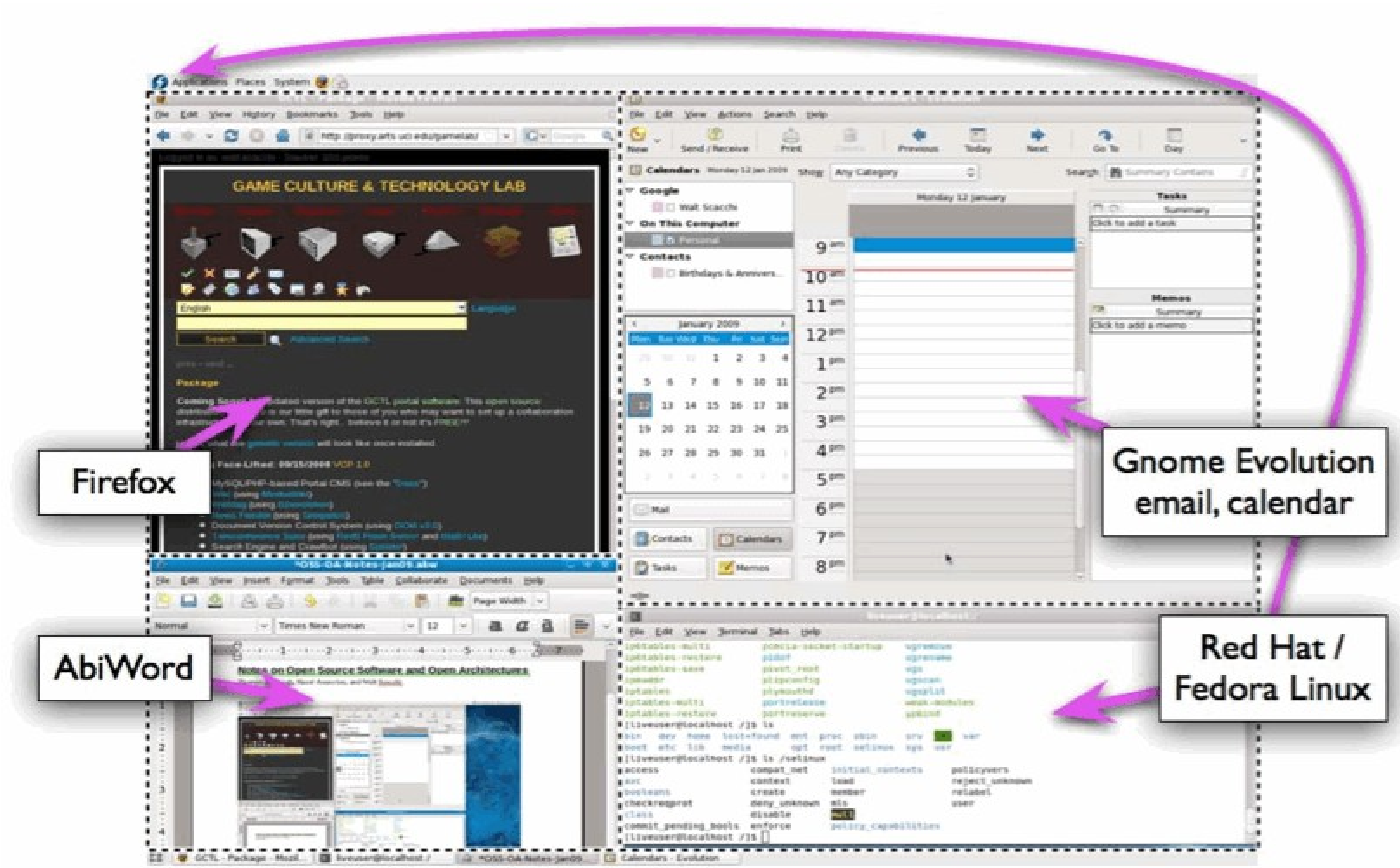
Build-time view of OA design selecting *proprietary* product family alternatives



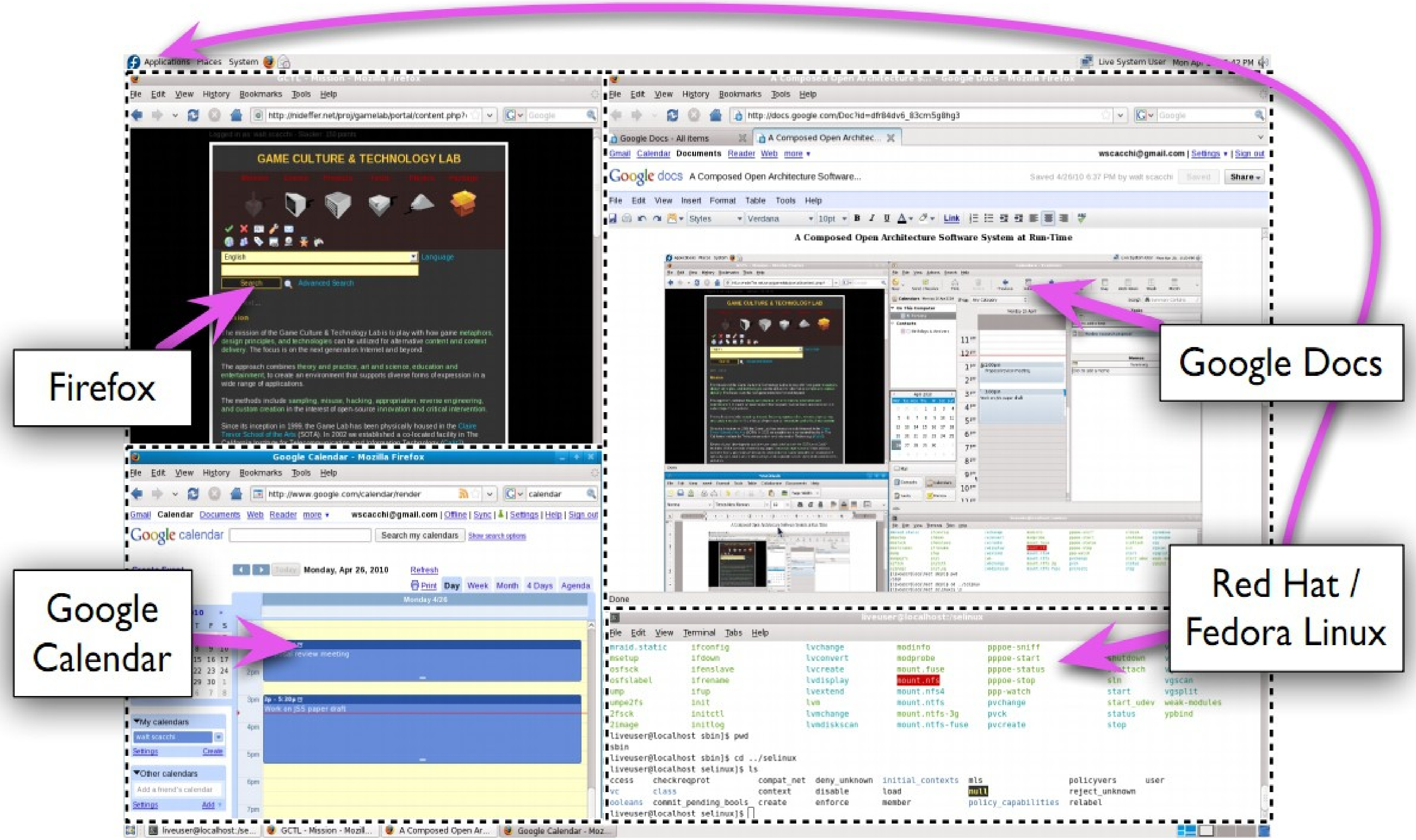
Build-time view of an OA system security encapsulation scheme



Run-time deployment view of OA system family member configuration



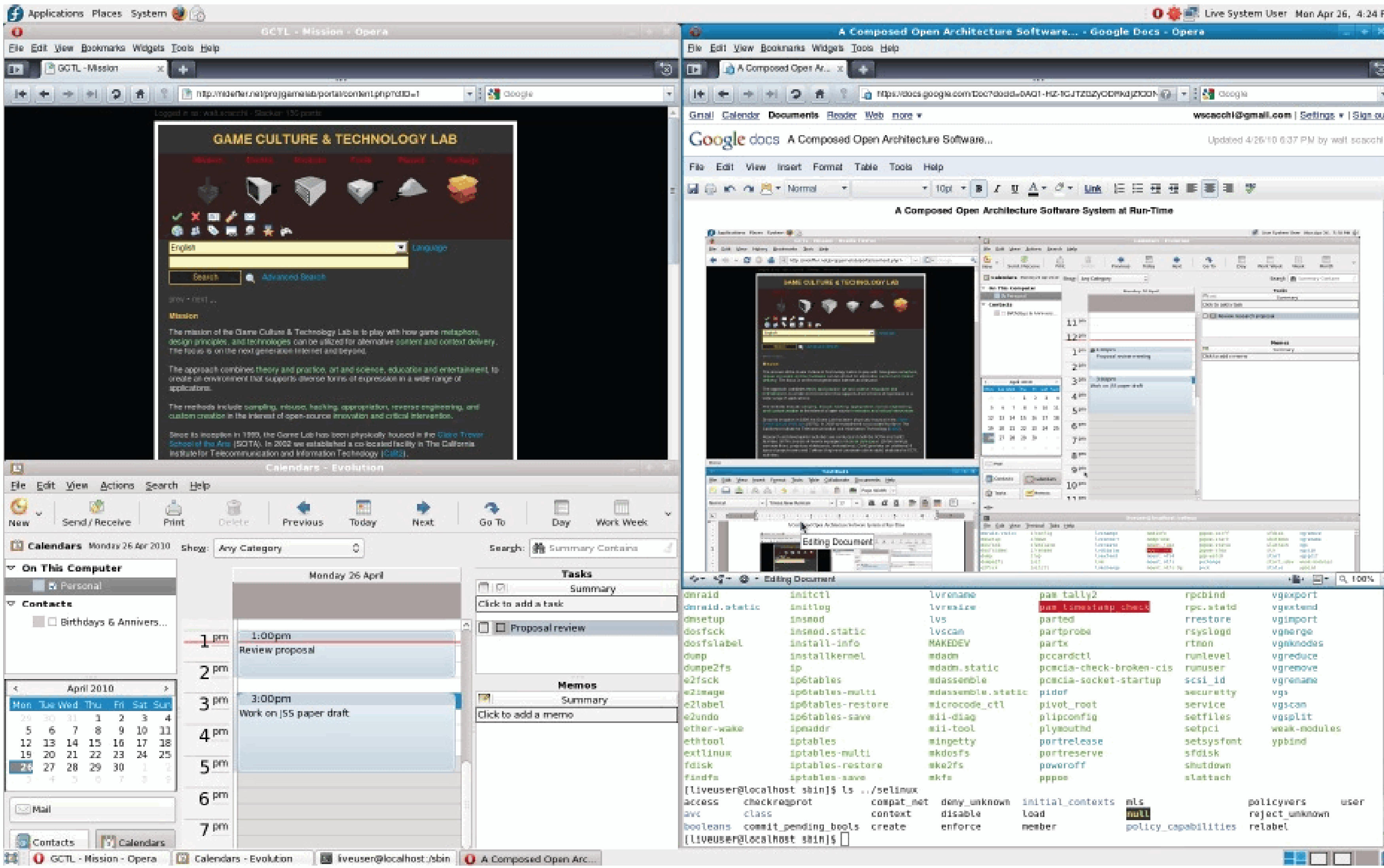
Run-time deployment view of a similar alternative OA system configuration



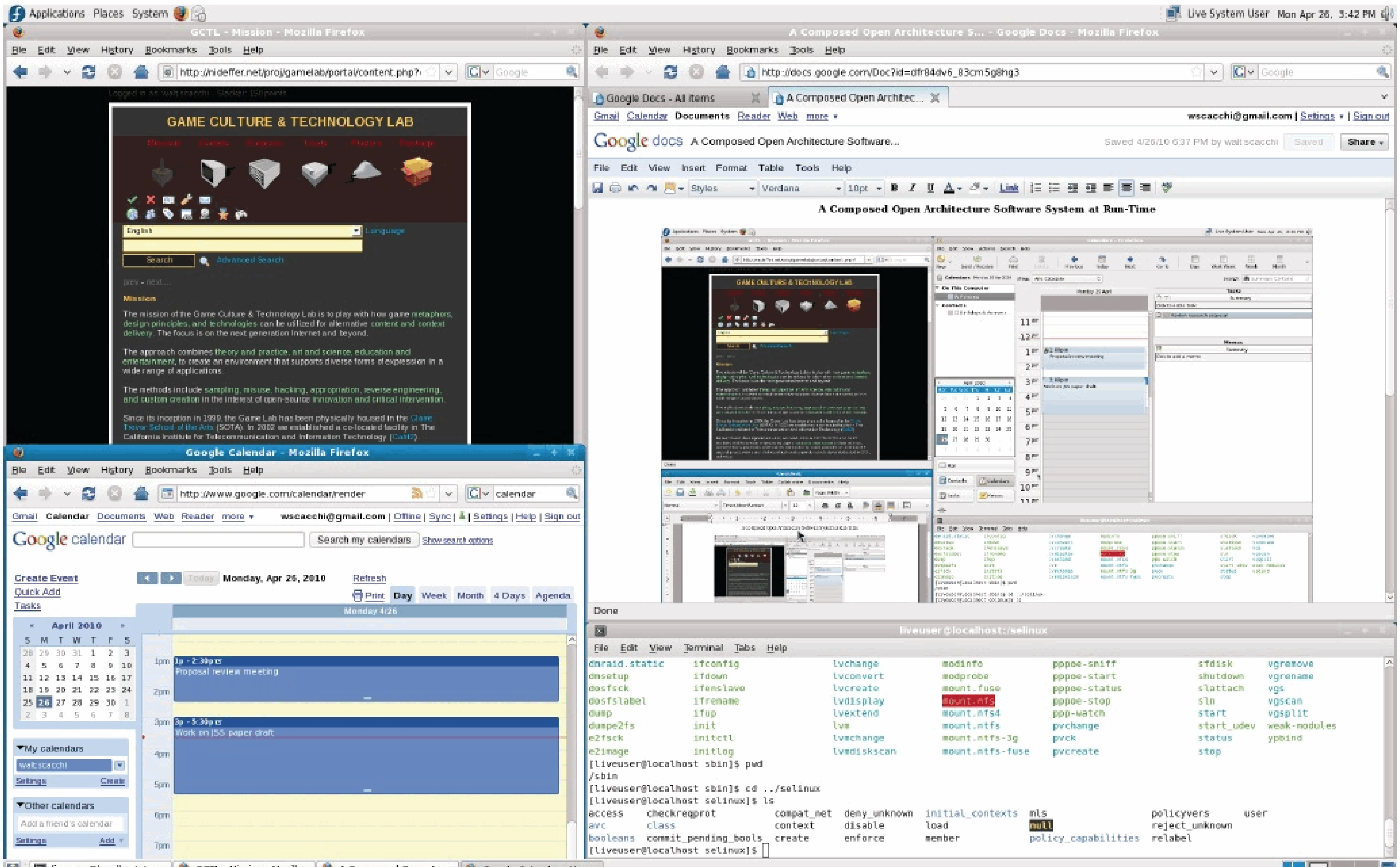
Types of evolutionary changes in OA systems that also change system configurations

- Component (version) evolution
- Component replacement by similar alternative
- Architecture evolution
  - Including dynamic reconfiguration
- Component security policy license evolution
  - Licenses represent *annotated constraints* on OA components, connectors, and configurations
- Change in desired license rights or acceptable obligations within an OA system

Run-time deployment view with alternative OA configuration



Run-time deployment view with service-based OA configuration



Combinatorially large space of alternative OA system configurations

- Example: OA system with 6 component types
  - Each component type defines a family of functionally similar alternatives
  - Assume three overall system platform alternatives
  - Each component type with 5 alternative producers
  - Each alternative providing 5 available versions
  - Each version providing 5 functionally equivalent variants
  - This allows for 6*3*5*5*5 = 2250 similar but distinct OA system configurations available for at build-time.

Combinatorially large space of alternative OA system configurations

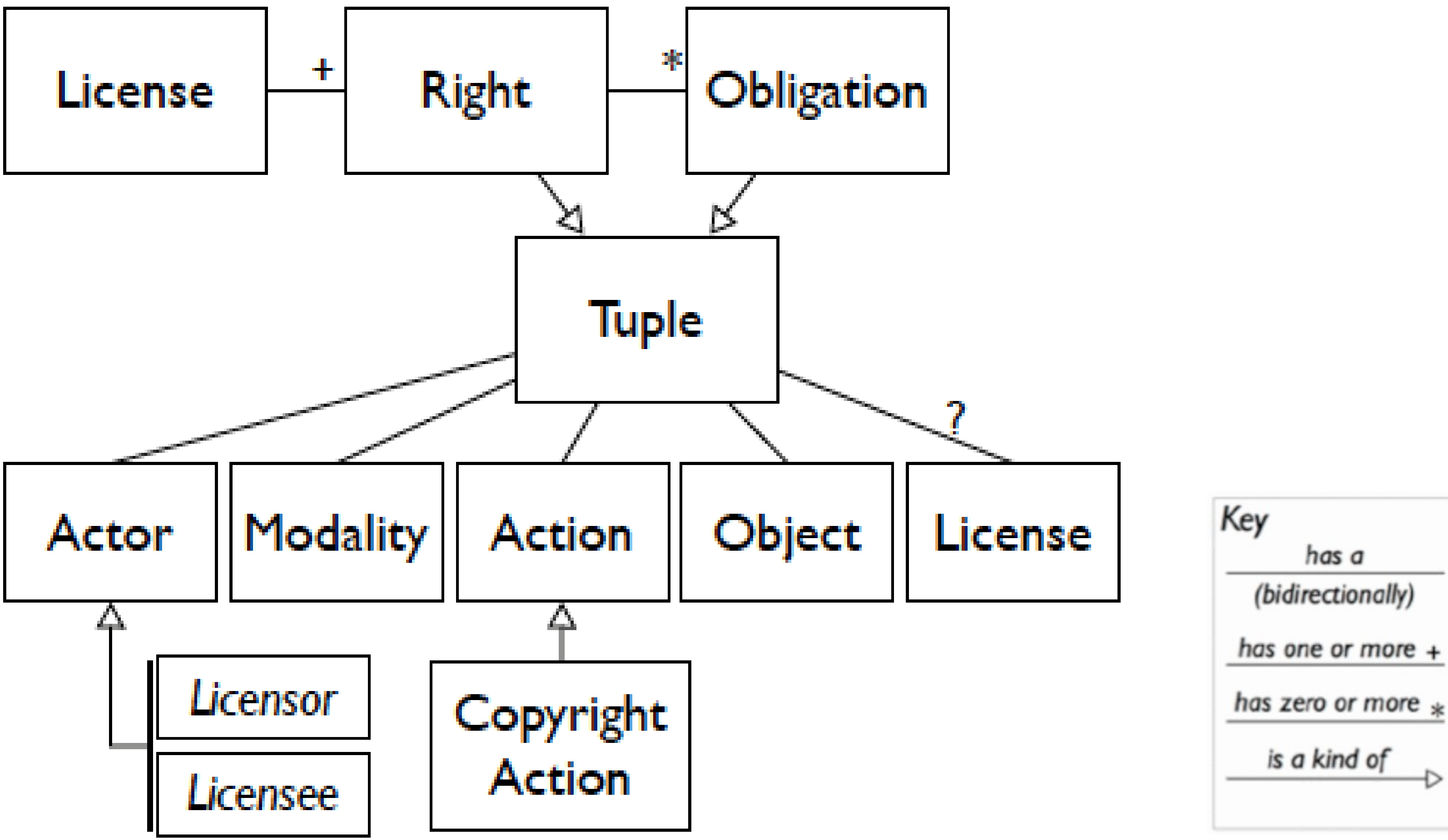
- Each configuration represents a specific attack surface, so *cost of attack grows combinatorially* with number of system alternatives to attack (i.e., which is the target now?)
- Cost of system defense via dynamic reconfiguration is low/constant
- Switching to alternative configurations can be handled via automated processes, driven by policy, e.g.:
  - switch run-time configuration every 30 minutes;
  - provide concurrent users with similar system configurations
  - monitor and continuously cross-check different user configurations for problem (attack or corrupted) operation
    - If configuration is problematic, then randomly switch to alternative similar configuration
    - If configuration is OK, then switch to equivalent alternative configuration at start of new usage sessions.

Software security licenses, architectures, and analysis

Specifying and analyzing system security requirements as "licenses"

- Security policies imply capabilities that correspond to "rights and obligations" in licenses
- Should be possible to specify and analyze system *security architecture* that conform to a *security meta-model*, much like we do for software licenses
- Should be possible to develop computational tools and development environments that can analyze security at design-time, build-time, and run-time, as well as when the system evolves

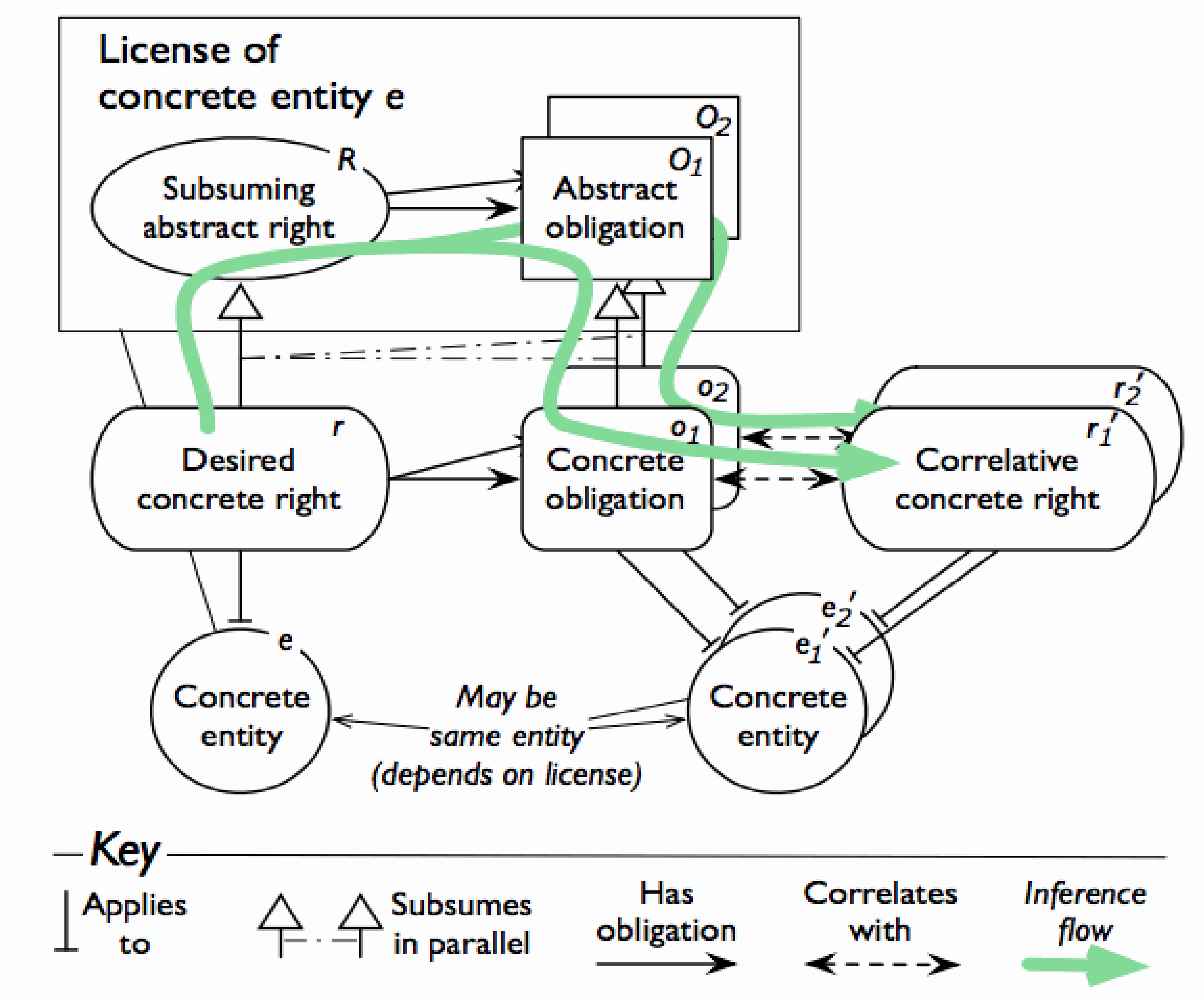
Software license meta-model for specifying constraint annotations



Logical modality and objects of software license rights and obligations constraints

<table>
<tr><th></th><th>Actor</th><th>Modality</th><th>Action</th><th>Object</th><th>License
(optional)</th></tr>
<tr><td rowspan="3">Abstract Right</td><td rowspan="9">Licensee
or
Licensor</td><td rowspan="4">May
or
Need Not</td><td rowspan="9">The set of actions is large, comprising whatever actions the licenses in question utilize</td><td>Any Under This License</td><td rowspan="3">This License
or
Object's License</td></tr>
<tr><td>Any Source Under This License</td></tr>
<tr><td>Any Component Under This License</td></tr>
<tr><td>Concrete Right</td><td rowspan="2">Concrete Object</td><td rowspan="2">Concrete License</td></tr>
<tr><td>Concrete Obligation</td><td rowspan="5">Must
or
Must Not</td></tr>
<tr><td rowspan="4">Abstract Obligation</td><td>Right's Object</td><td rowspan="4">Concrete License
or
Right's License</td></tr>
<tr><td>All Sources Of Right's Object</td></tr>
<tr><td>X Scope Sources</td></tr>
<tr><td>X Scope Components</td></tr>
</table>

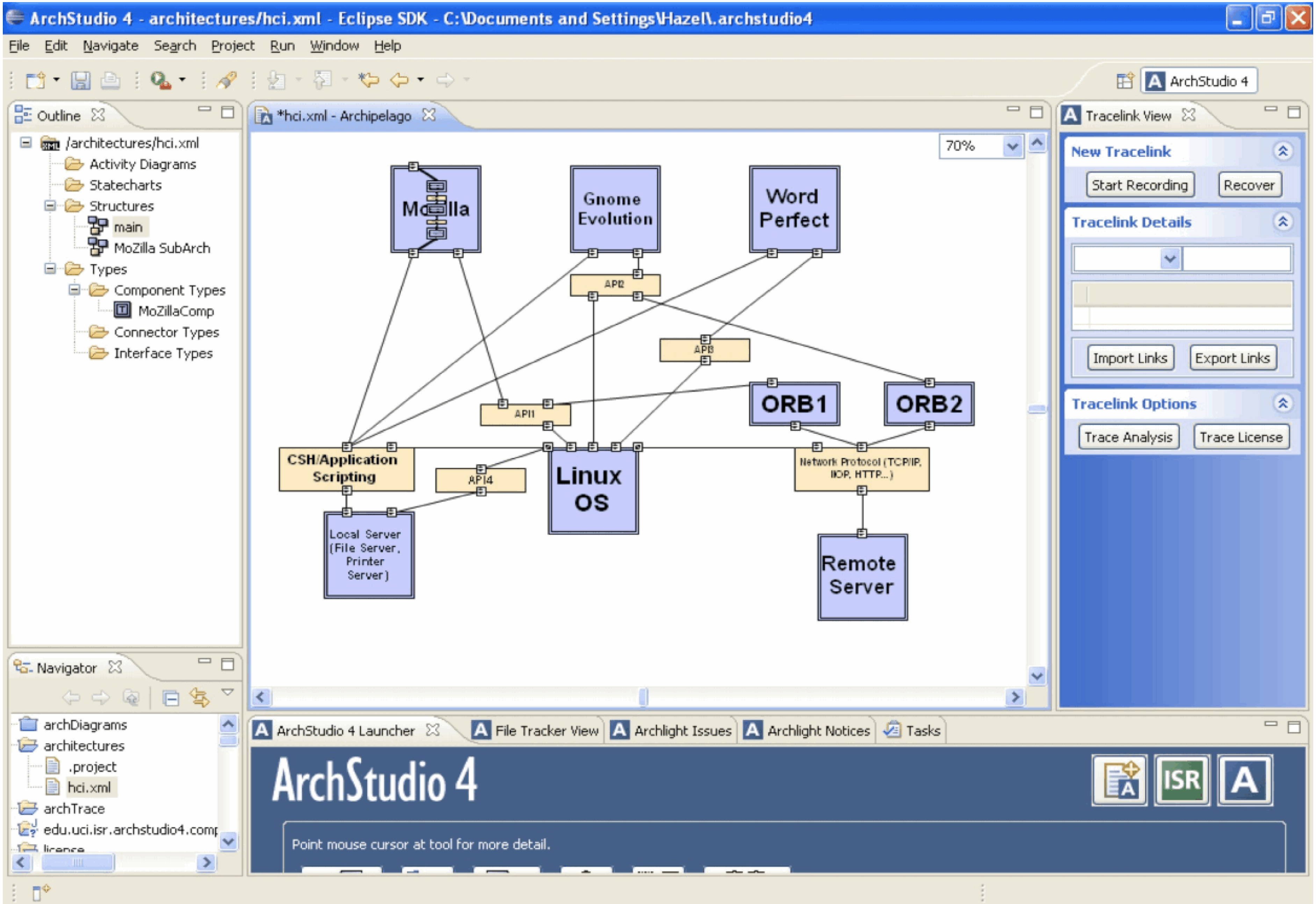
License inference scheme



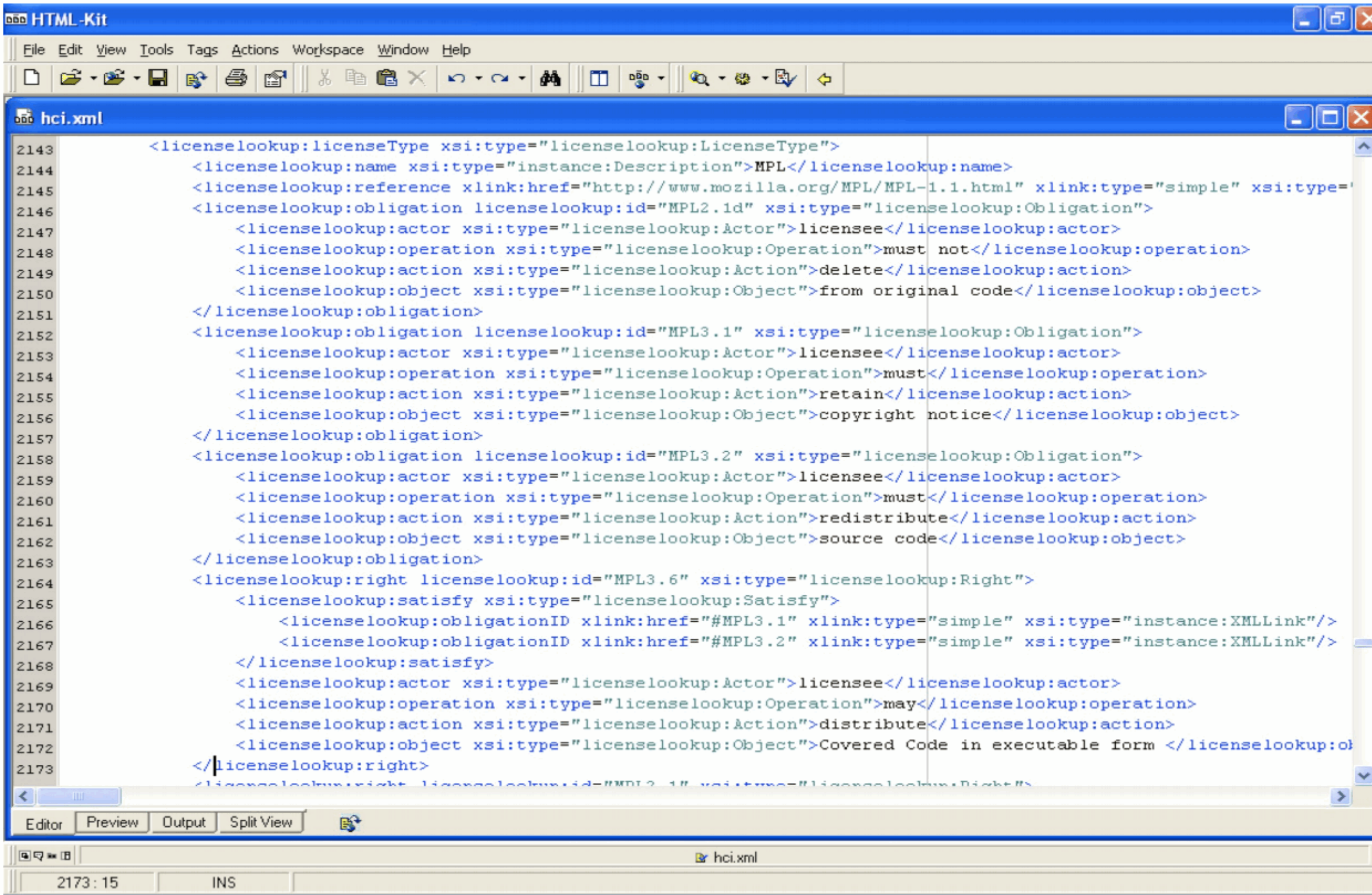
Security license analysis

- License types:
  - Strongly reciprocal, weakly reciprocal, academic, Terms of Service, Proprietary
- Propagation of reciprocal obligations
- Conflicting obligations
- Calculating obligations and rights

Prototype view of OA system development environment with license analysis plug-in



Internal form of component license annotation of OA prior to analysis



Directory of computational methods for analyzing “rights”

RightConcrete (Thomas' Packages)

file:///Users/thomas/java/javadoc/index.html

Reader

All Classes

Packages
automaton
bibtex
commands
eventmatch
fologic
lambda
license0
license3
nesting
org.thomasalspaugh.command

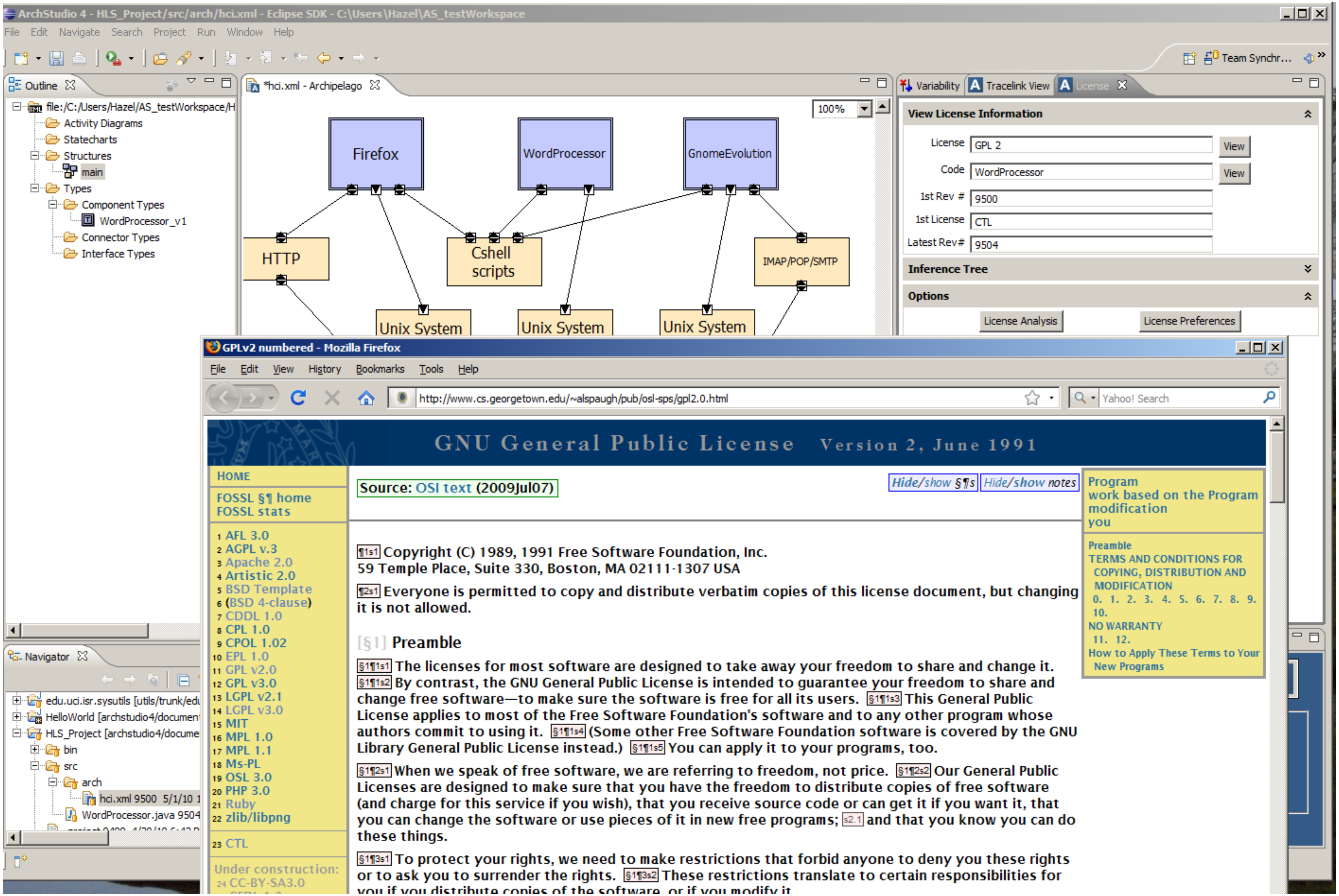
Obligation
PartiallyOrdered
Patient
Right

Classes
Action
Actor
Calculation
Command
Domain
DomainScope
DomainSet
FormulaConjunction
FormulaDisjunction
LicenseAbstract
LicenseConcrete
LicenseConcretePrecursor
LicenseQuantifier
LicenseReference
LicenseSet
LicenseVirtual
Modality
ModalityObligation
ModalityRight
ObligationAbstract
ObligationConcrete

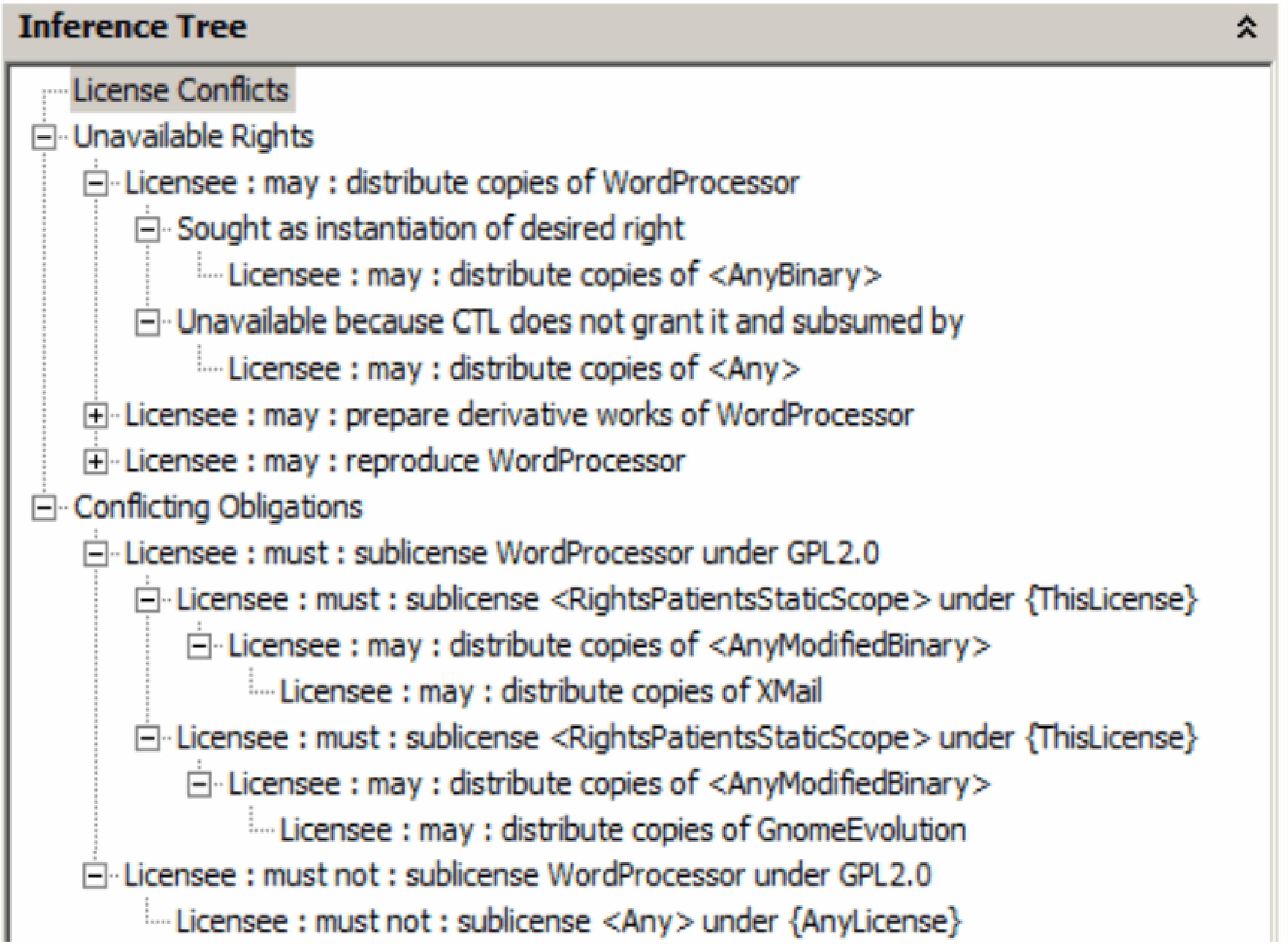
Method Summary

Action	**action**() The right's action.
Actor	**actor**() The right's actor.
int	**compareTo**(Named _o) Compares this object with the specified object in a total ordering by name().
boolean	**conflictsWith**(ObligationAbstract _oa) True if this right conflicts with _obligation.
ObligationConcrete	**correlative**() The correlative obligation.
static RightConcrete	**factory**(Actor _actor, ModalityRight _modality, Action _action, PatientConcrete _patient, LicenseConcrete _license) Returns the concrete right that has the given arguments, constructing it if necessary.
static RightConcrete	**factory**(RightAbstract _abstractRight, PatientConcrete _context) Returns a concrete right subsuming an abstract right.
LicenseConcrete	**license**() The right's license argument.
Modality	**modality**() The right's modality (a ModalityRight).
String	**name**() The name (same as toString()) used in comparisons.
protected boolean	**overlapsTuple**(license3.Tuple _b) True iff subsumes(_b) or _b.subsumes(this).
PatientConcrete	**patient**() The right's patient argument.
boolean	**reserved**() True if this right overlaps with a reserved right.
RightAbstract	**reservedRight**()

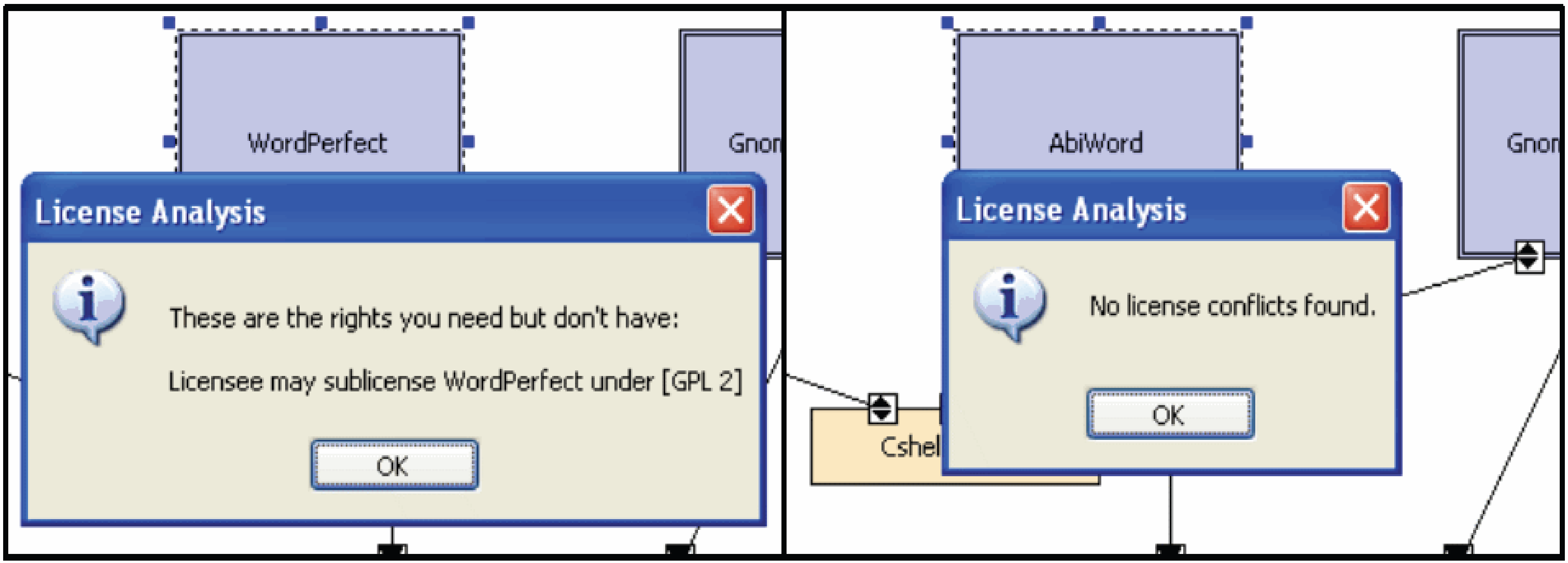
License review during license analysis



Reasoning structure during analysis



Results from license analyses with system component replacement



Discussion

- Cyber security challenges of interest
  - Automated reasoning about OA system security via robust and resilient autonomic host services
    - System self-awareness
    - Self-diagnosis (static/dynamic monitoring)
    - Self-healing (dynamic OA reconfiguration)
    - Supporting graceful degradation and artificial diversity
- Reasoning about secure Open Architecture system obligations and rights
- Examples of recent or work-in-progress results

Acknowledgements

This presentation was developed under work supported by the Naval Postgraduate School Acquisition Research Program Assistance Agreement No. #N00244-10-1-0038, #N00244-12-1-0004 and #N00244-12-1-0067 awarded by the U.S. Fleet Industrial Supply Center, San Diego (FISC-SD). The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the U.S. FISC-SD and NPS. The FISC-SD and NPS do not endorse any products or commercial services mentioned in this publication.

Funding also from grants #0808783 and #1256593 from the National Science Foundation. No review, approval, or endorsement implied.