Model-driven Engineering for Agile Development & Integration of Large-scale Component-based Systems

http://www.dre.vanderbilt.edu/CUTS/MoDELS08.pdf

James H. Hill & Aniruddha Gokhale
{j.hill, a.gokhale}@vanderbilt.edu

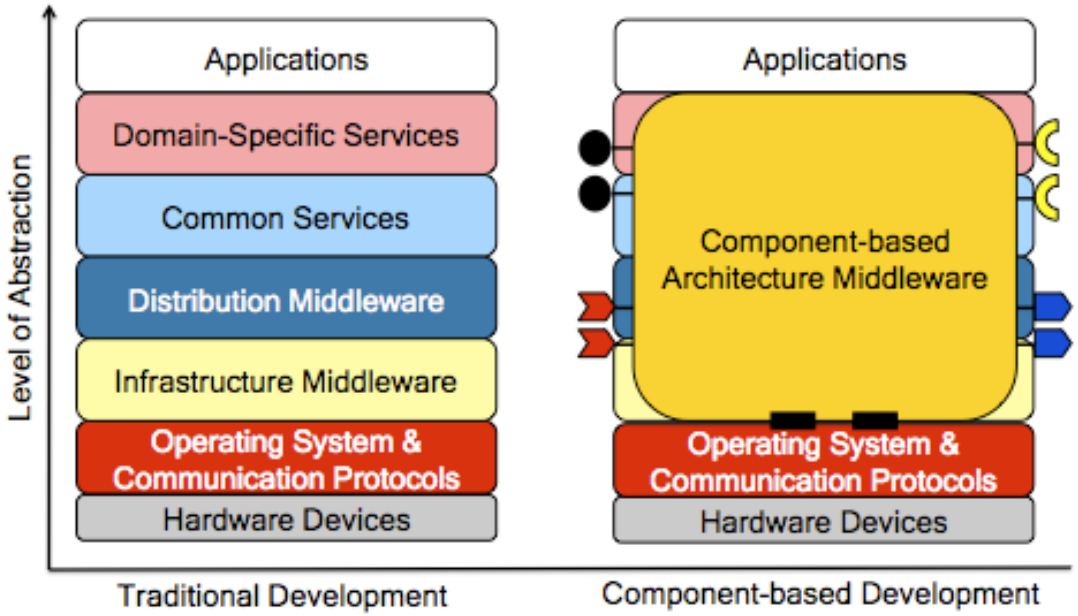
Institute for Software Integrated Systems
Dept of Elec Eng & Comp Sc
Vanderbilt University
Nashville, Tennessee, USA

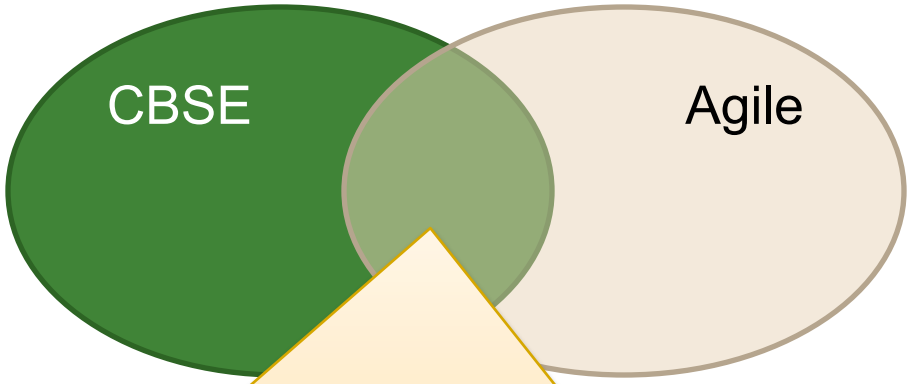
Tutorial Outline

- Tutorial Synopsis
- Agile Development & Component-based Software Engineering for Distributed Component-based Systems
- MDE Fundamentals & Applicability to Distributed Component-based Systems
- Coffee Break
- Early System Integration Testing
- Hands-on Demo with CUTS
- Continuous System Integration
- Demo with CiCUTS
- Deployment-based Analysis (optional)

Tutorial Synopsis

- Traditional methods of developing distributed real-time & embedded (DRE) systems are being replaced by component-base architecture middleware
  - *i.e.*, component-based software engineering (CSBE)
- Agile development is an software engineering paradigm aimed to address the challenges of rapid software development





Tutorial Goal: Combine agile software techniques with CBSE to address the quality of service (QoS) needs of enterprise DRE systems

PART 1:
Agile Development & Component-based Software Engineering for Distributed Component-based Systems

Context: Component-based Software Engineering (CBSE)

Enables the rapid development & deployment of enterprise, distributed systems via system composition & built-in lifecycle services

Standardized Component Models

- CORBA Component Model (CCM)
- J2EE/EJB
- Microsoft .NET Web Services

Characteristics of CBSE

- Improves & expedites the software development process
- Promotes reuses of core-intellectual property through *development via composition*
- Shifts developer focus to application functionality

Manifesto for CBSE
Separation of concerns… Focus on "business-logic"… Reuse of intellectual property

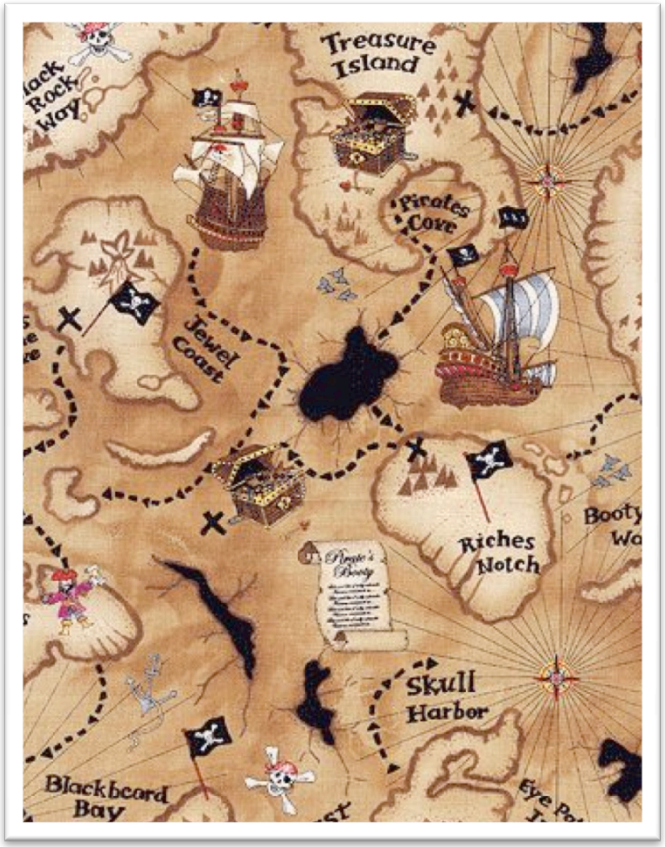
Context: Agile Development

Agile Software Development

- A process that embraces *small & lightweight* development techniques rather than a long & heavyweight development process

- Focuses on working software delivered *continuously* throughout the development lifecycle

- *Adaptable* to changes in software requirements



Characteristics of Agile Processes

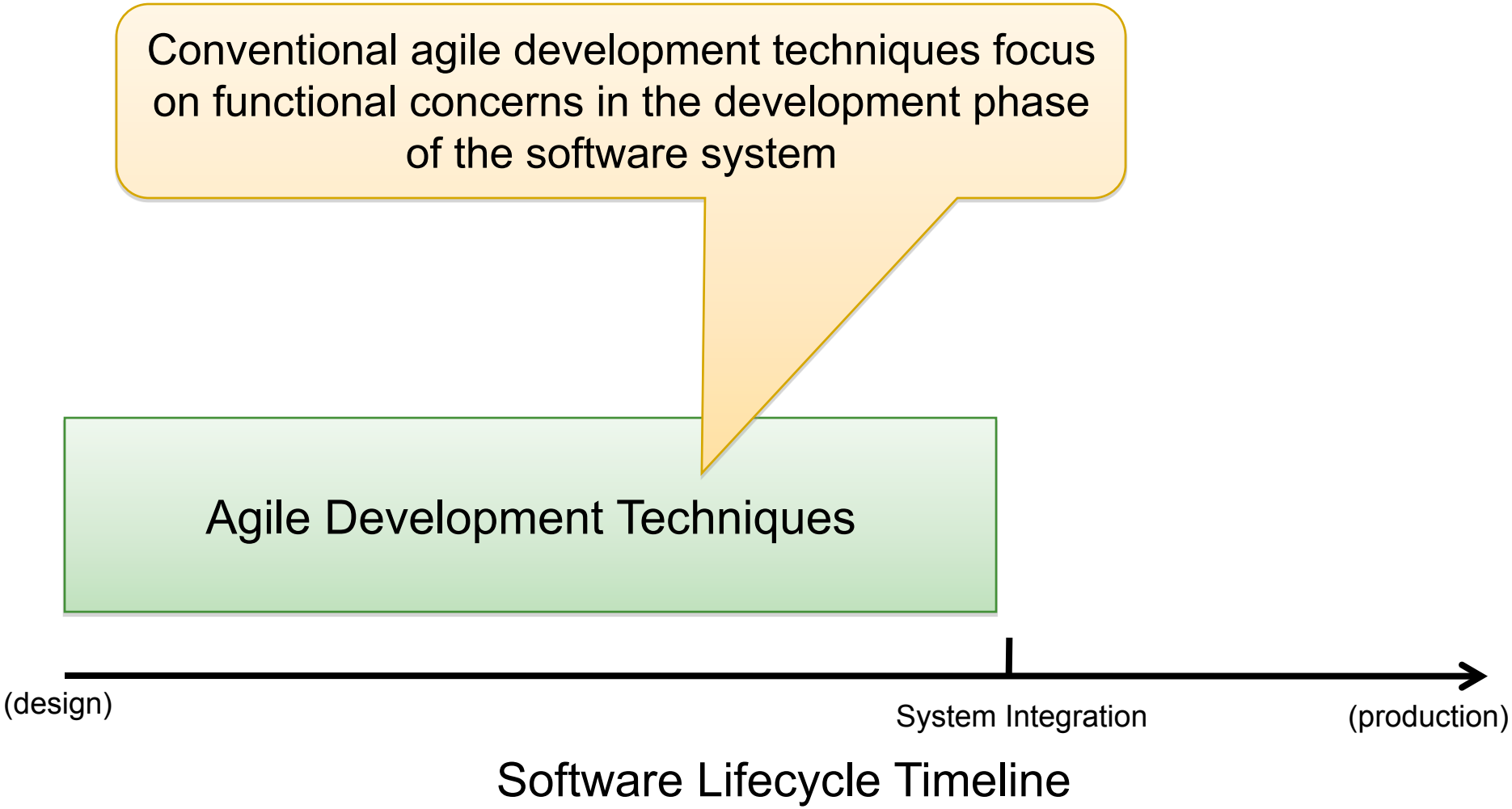
- Continuous Integration
- Test-driven Development
- Behavior-driven Development
- Scrum

Manifesto for Agile Development
Individuals & interactions… Working software… Customer collaboration… Responding to change…

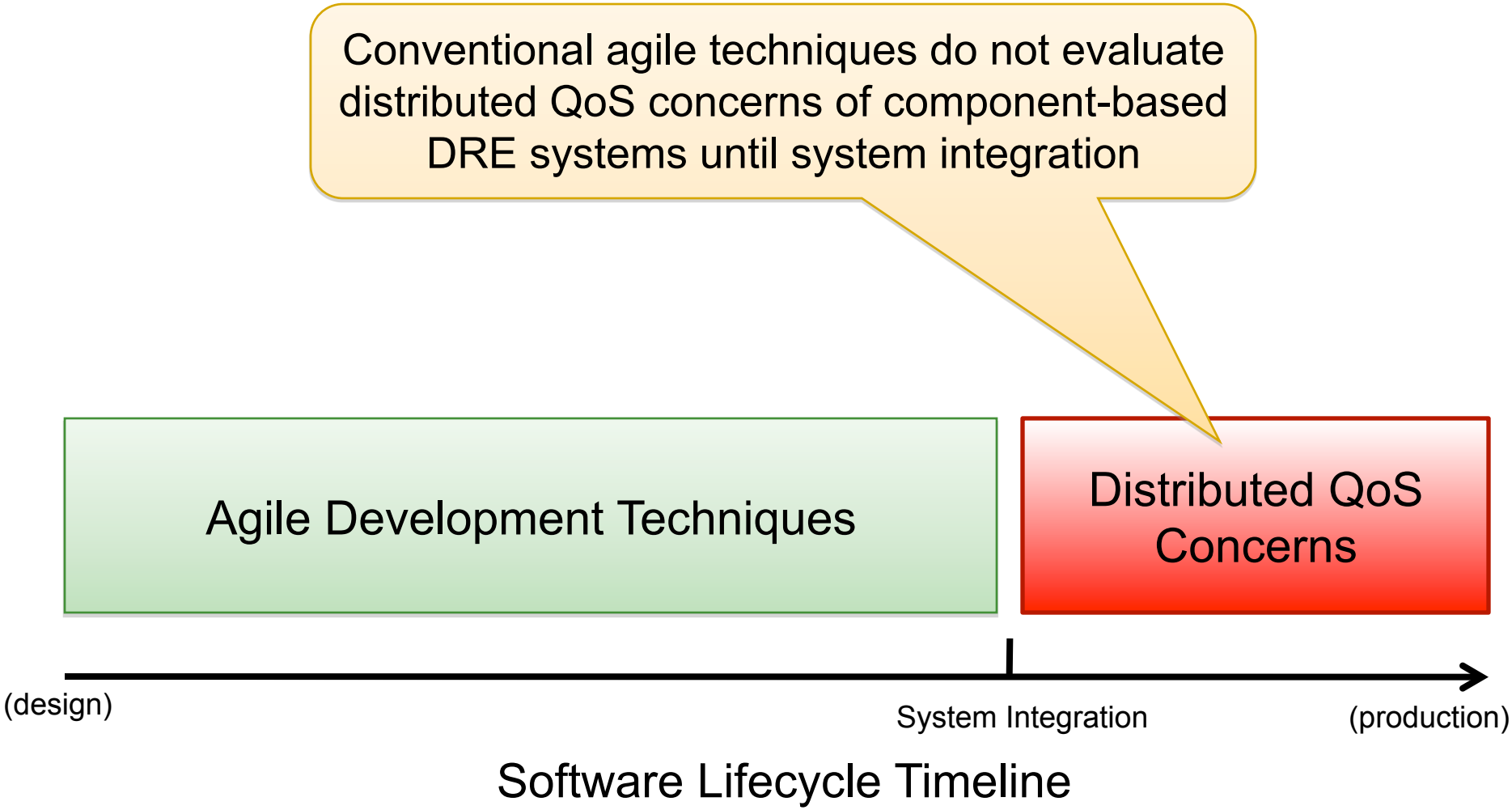
Summary of Key Agile Development Techniques

Technique	Description/Characteristics
Test-Driven Development (TDD)	• Write test case first, then implement functionality to pass the test case • Additional functionality *should not* break existing test cases
Behavior-Driven Development (BDD)	• Complement of test-driven development • Create test scenarios that describe behavior of software; then implement functionality to pass the scenario
Continuous Integration (CI)	• Continuously build & evaluate software functionality throughout development lifecycle

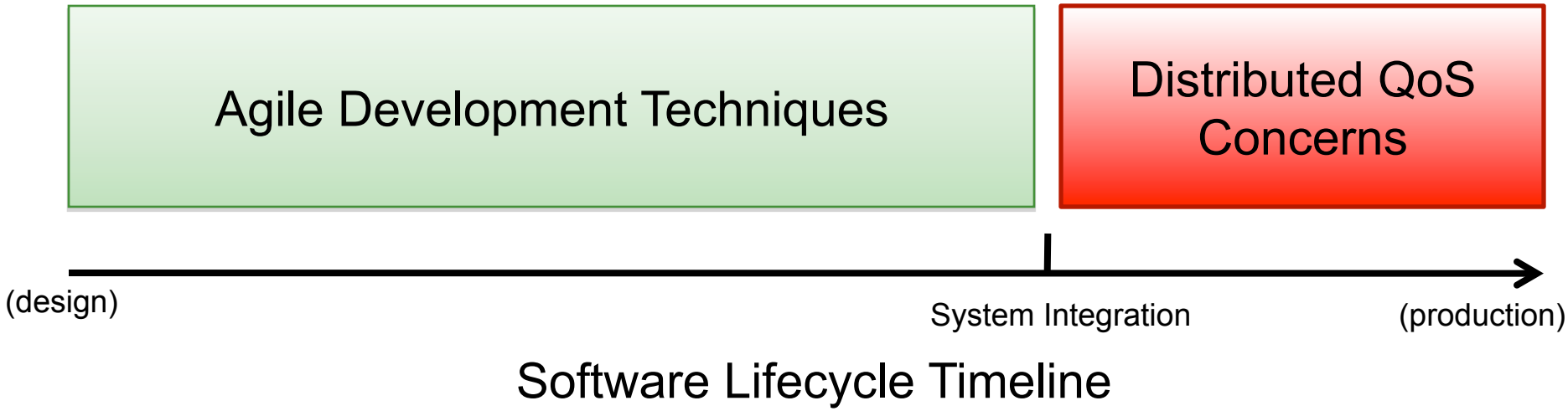
Combining Agile & CBSE for DRE Systems: Complexity #1



Combining Agile & CBSE for DRE Systems: Complexity #1

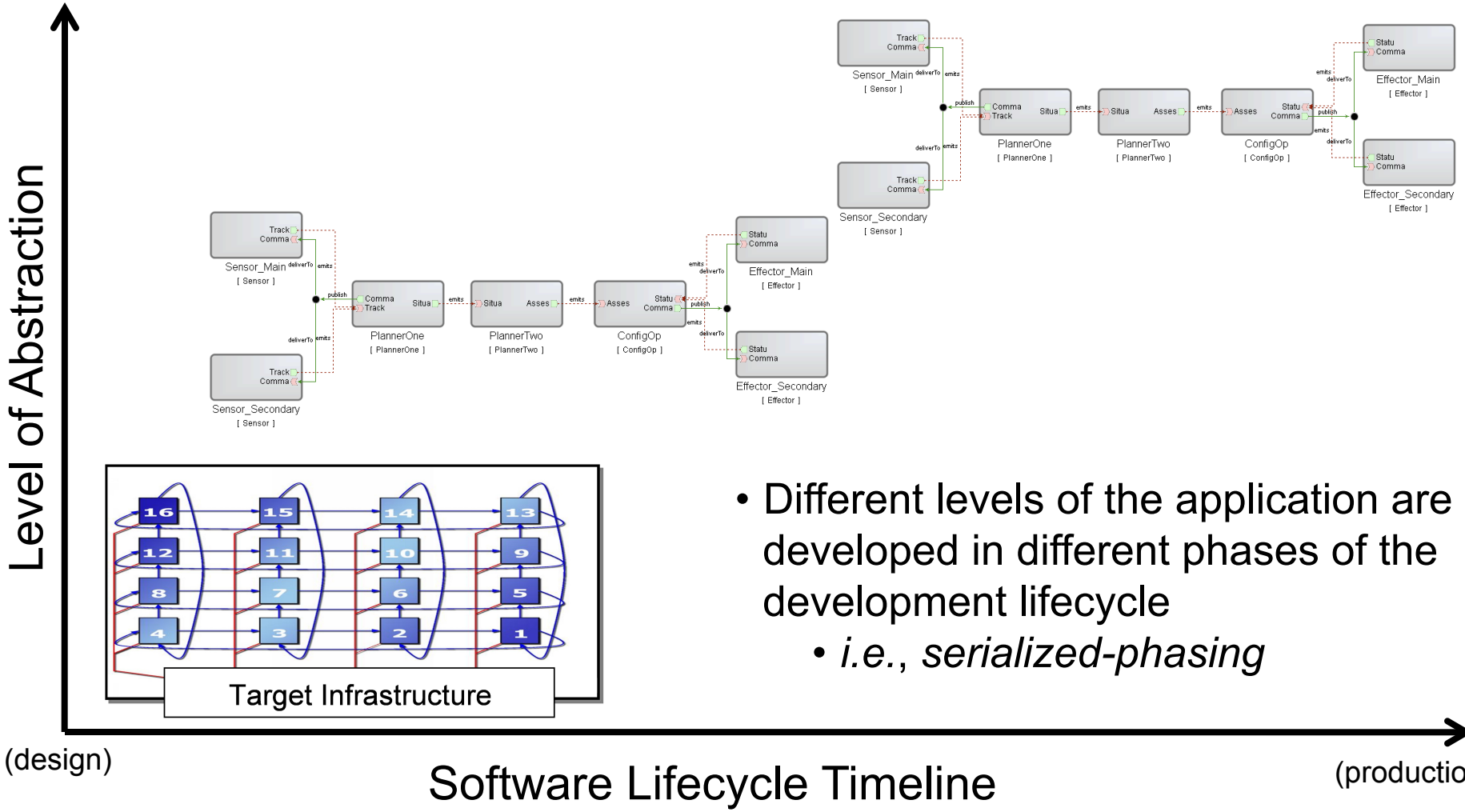


Combining Agile & CBSE for DRE Systems: Complexity #1



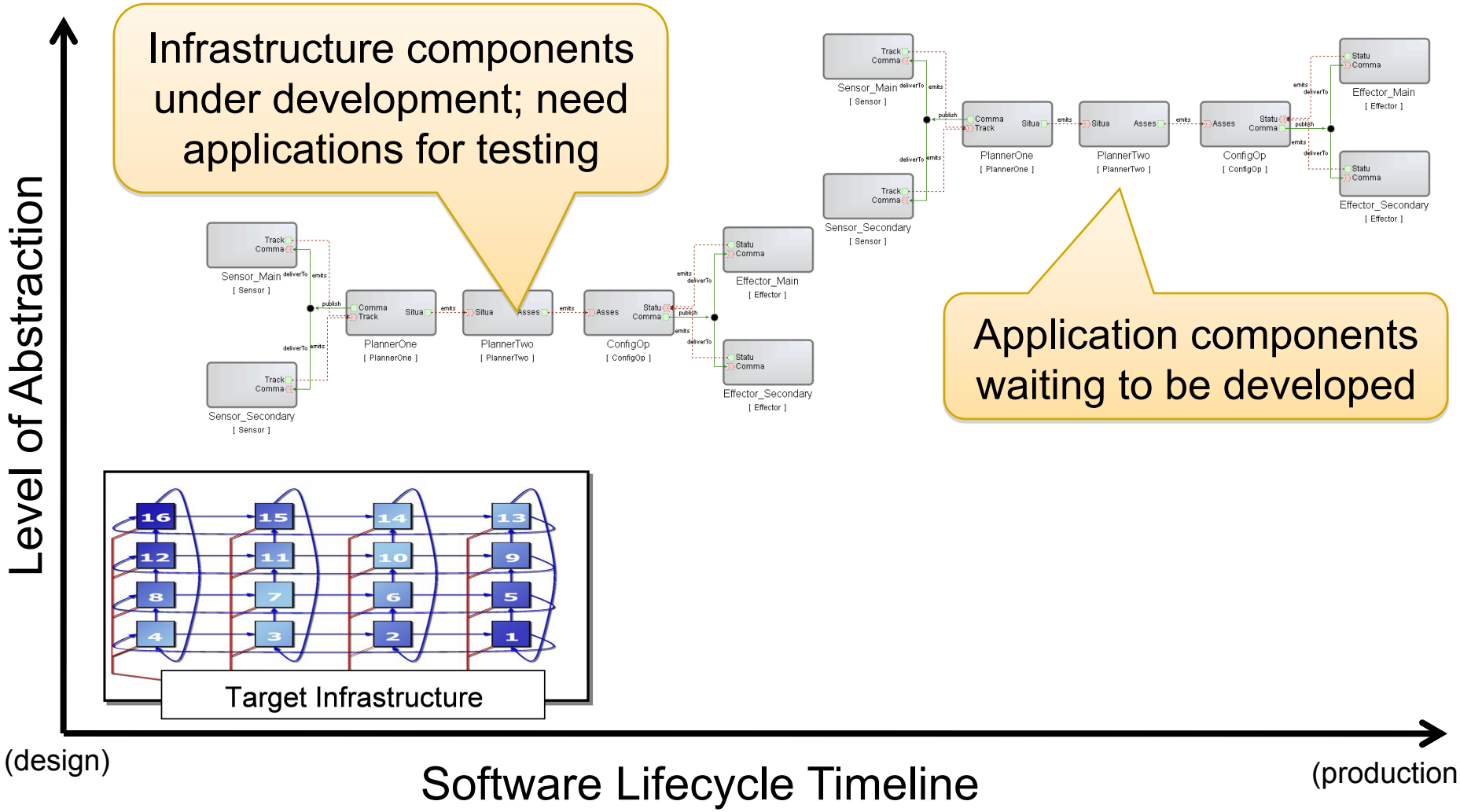
Disconnect between software development & software integration makes it hard to understand how each dimension affects the other in the software lifecycle

Combining Agile & CBSE for DRE Systems: Complexity #2

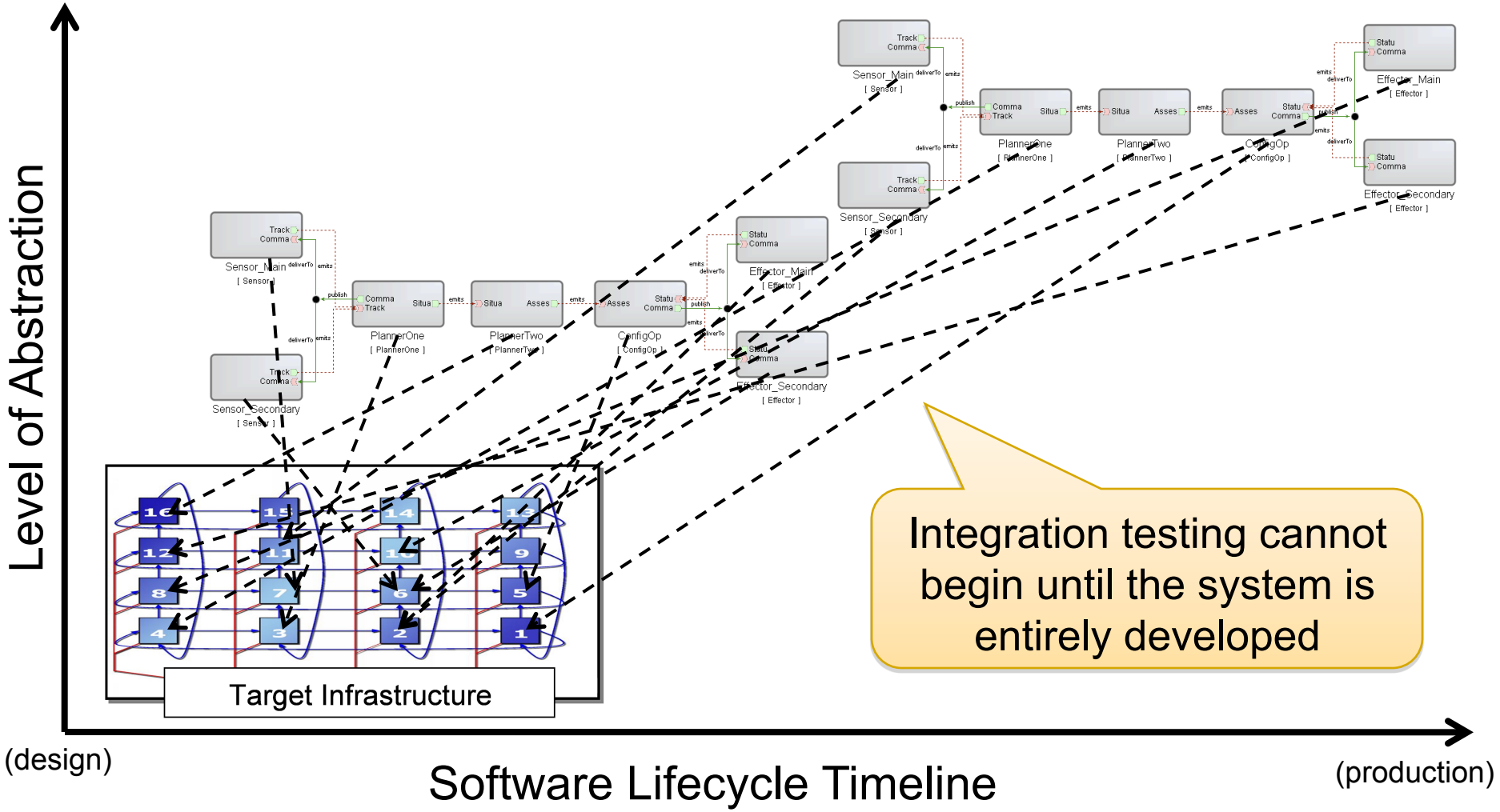


- Different levels of the application are developed in different phases of the development lifecycle
  - *i.e.*, *serialized-phasing*

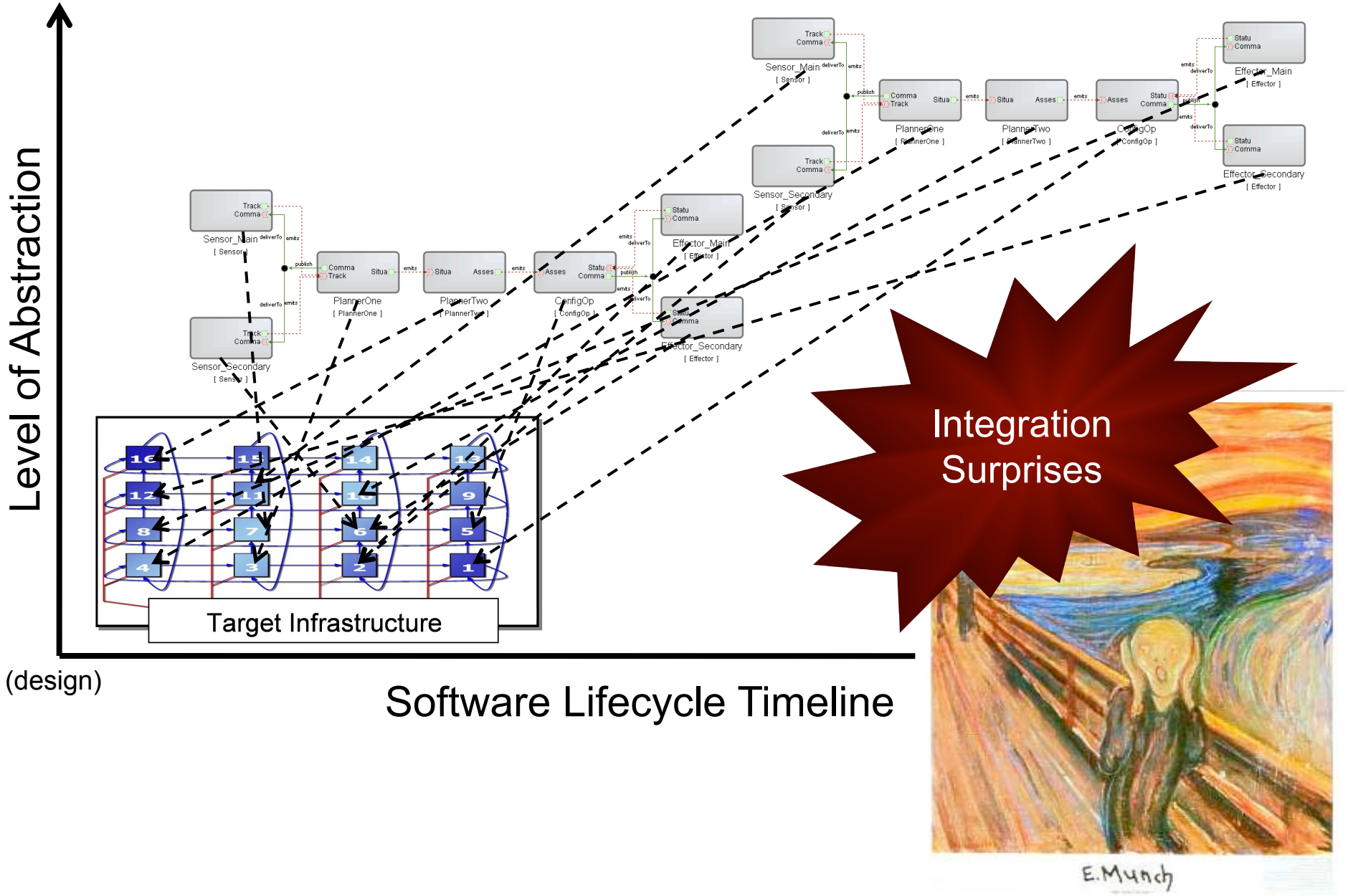
Understanding Serialized Phasing in CBSE (1/3)



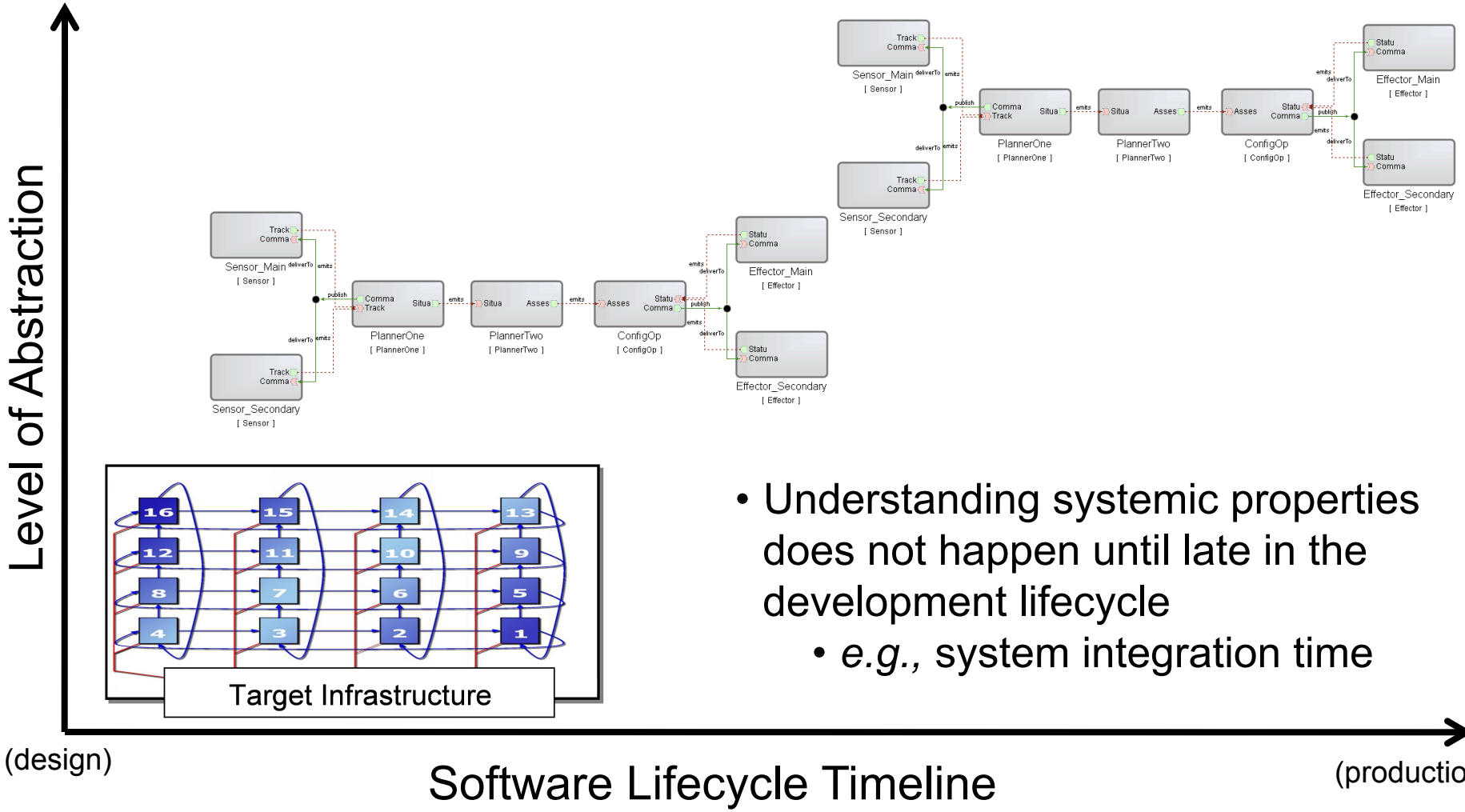
Understanding Serialized Phasing in CBSE (2/3)



Understanding Serialized Phasing in CBSE (3/3)



Combining Agile & CBSE for DRE Systems: Complexity #2

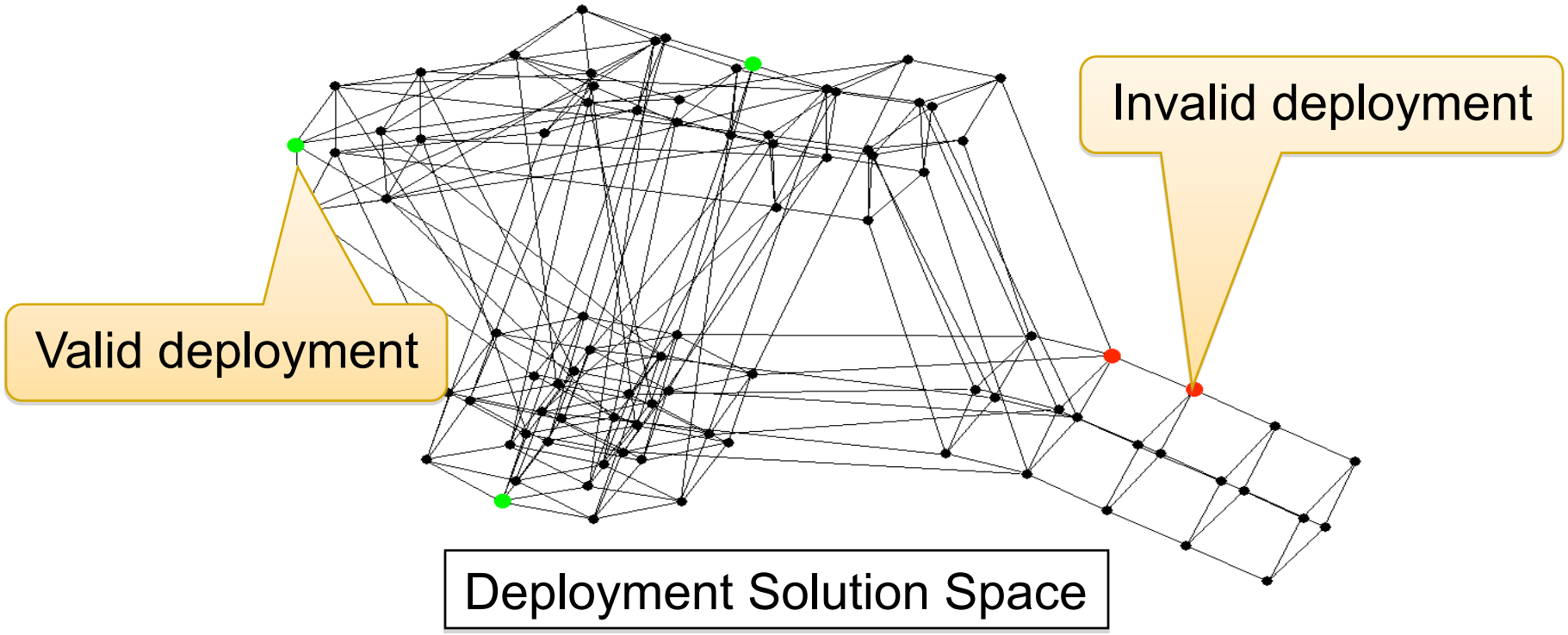


- Understanding systemic properties does not happen until late in the development lifecycle
  - *e.g.*, system integration time

Serialized-phasing of component-based DRE systems makes it hard to *continuously integrate* & understand distributed QoS concerns in software lifecycle

Combining Agile & CBSE for DRE Systems: Complexity #3

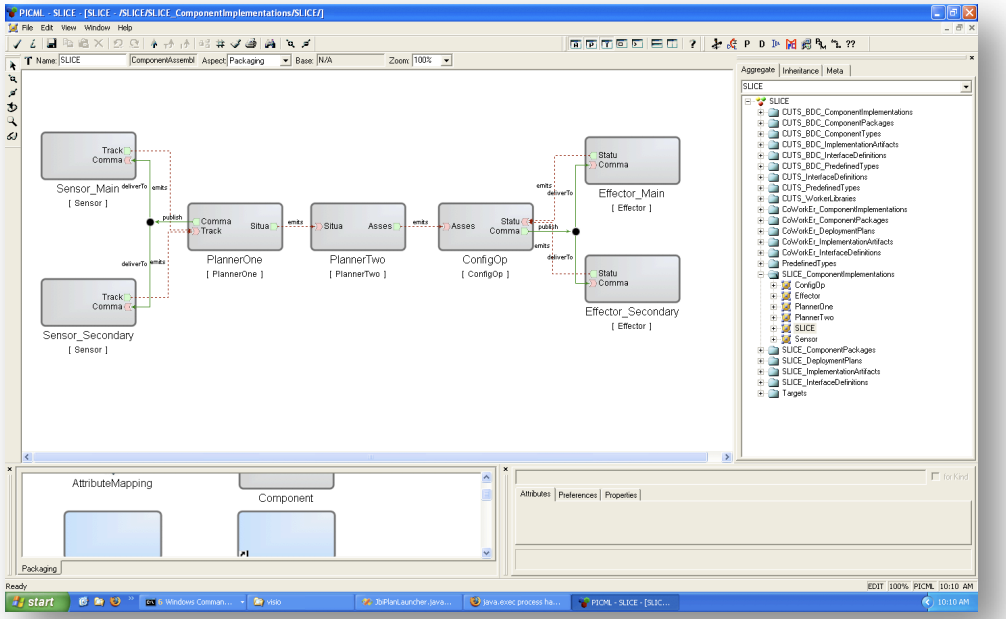
- Developers must test *many* deployment & configurations (D&Cs) to locate ones that meet QoS requirements



Key Challenges

- Component-based systems can be **deployed in many ways** (*e.g.*, $|host|^{|components|}$)
- **Emulating every deployment is costly**, *e.g.*, time spent running false deployments (or not running valid deployments) can translate to lost project dollars
- Locating valid deployments that meet performance requirements is a **decision problem,** *i.e.*, NP hard

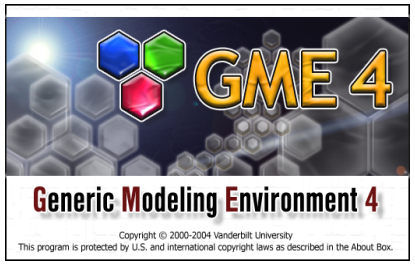
PART 2:
MDE Fundamentals & Applicability to Distributed Component-based Systems



Technology Enabler: Generic Modeling Environment (GME)

"Write Code That Writes Code That Writes Code!"

GME Architecture



Application Developers (Modelers)

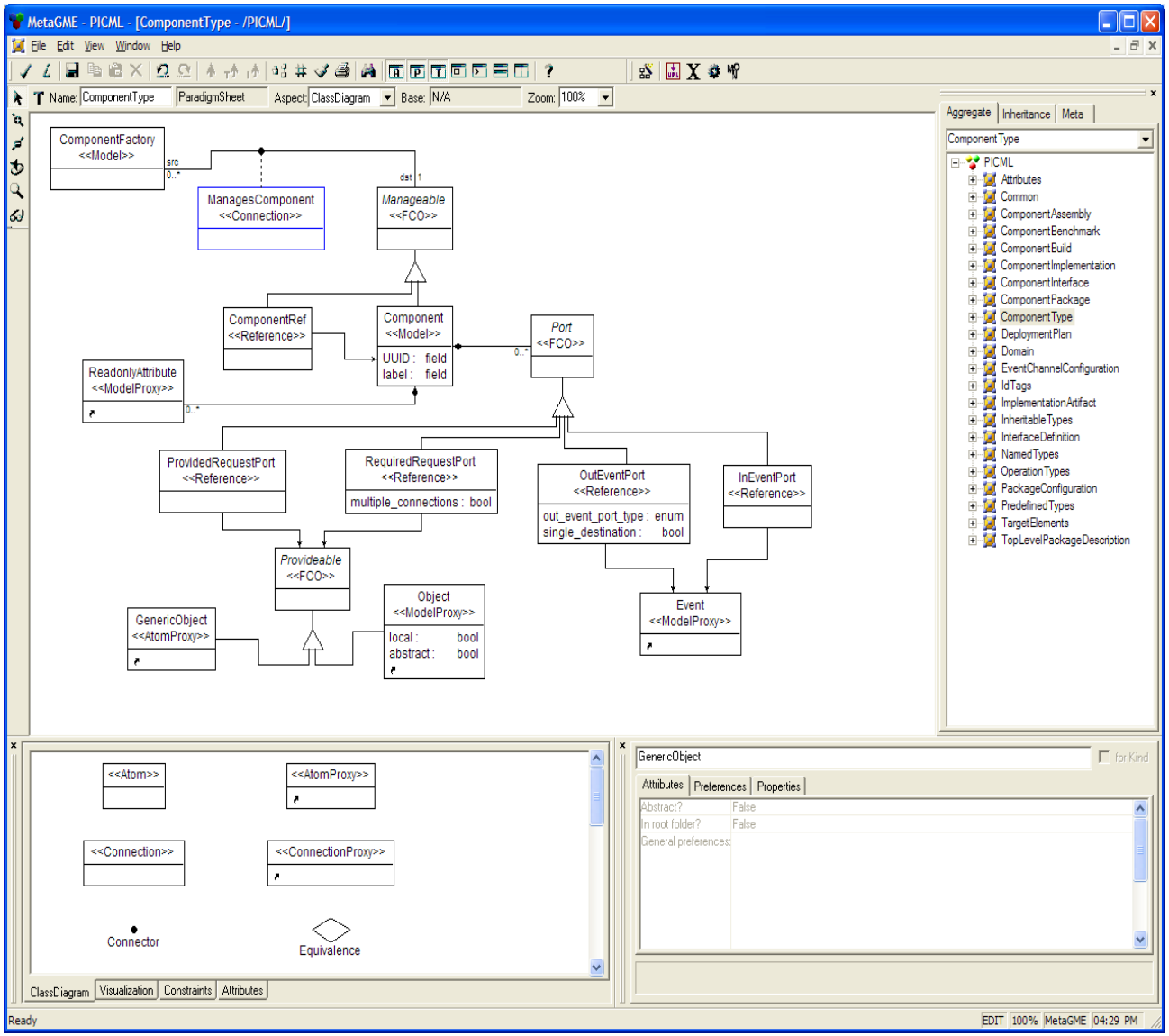
MDE Tool Developer (Metamodeler)

Goal: Correct-by-construction component-based systems

www.isis.vanderbilt.edu/Projects/gme/default.htm

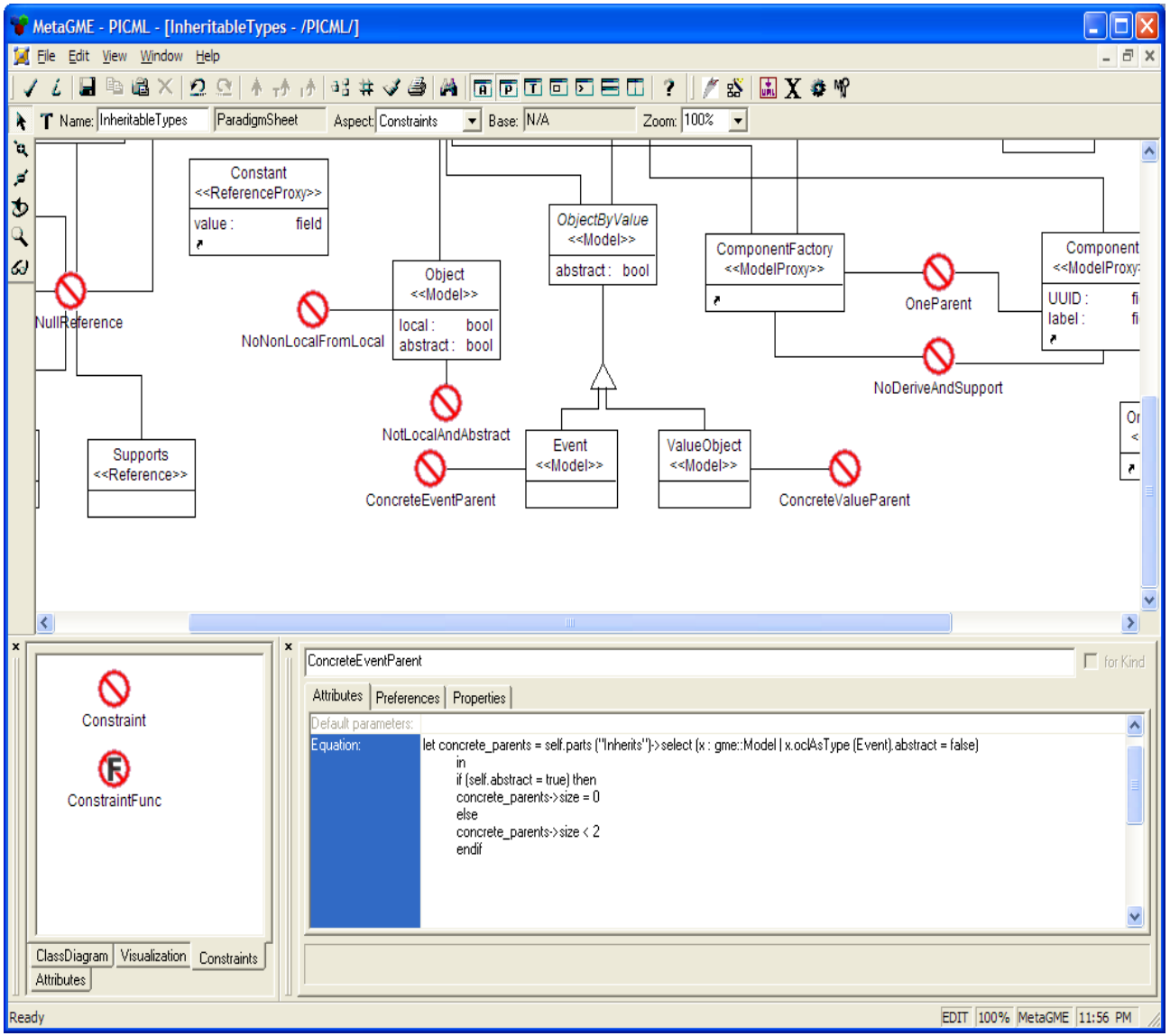
MDE Tool Development in GME

- Tool developers use MetaGME to develop a *domain-specific graphical modeling environment*
  - Define syntax & visualization of the environment via *metamodeling*



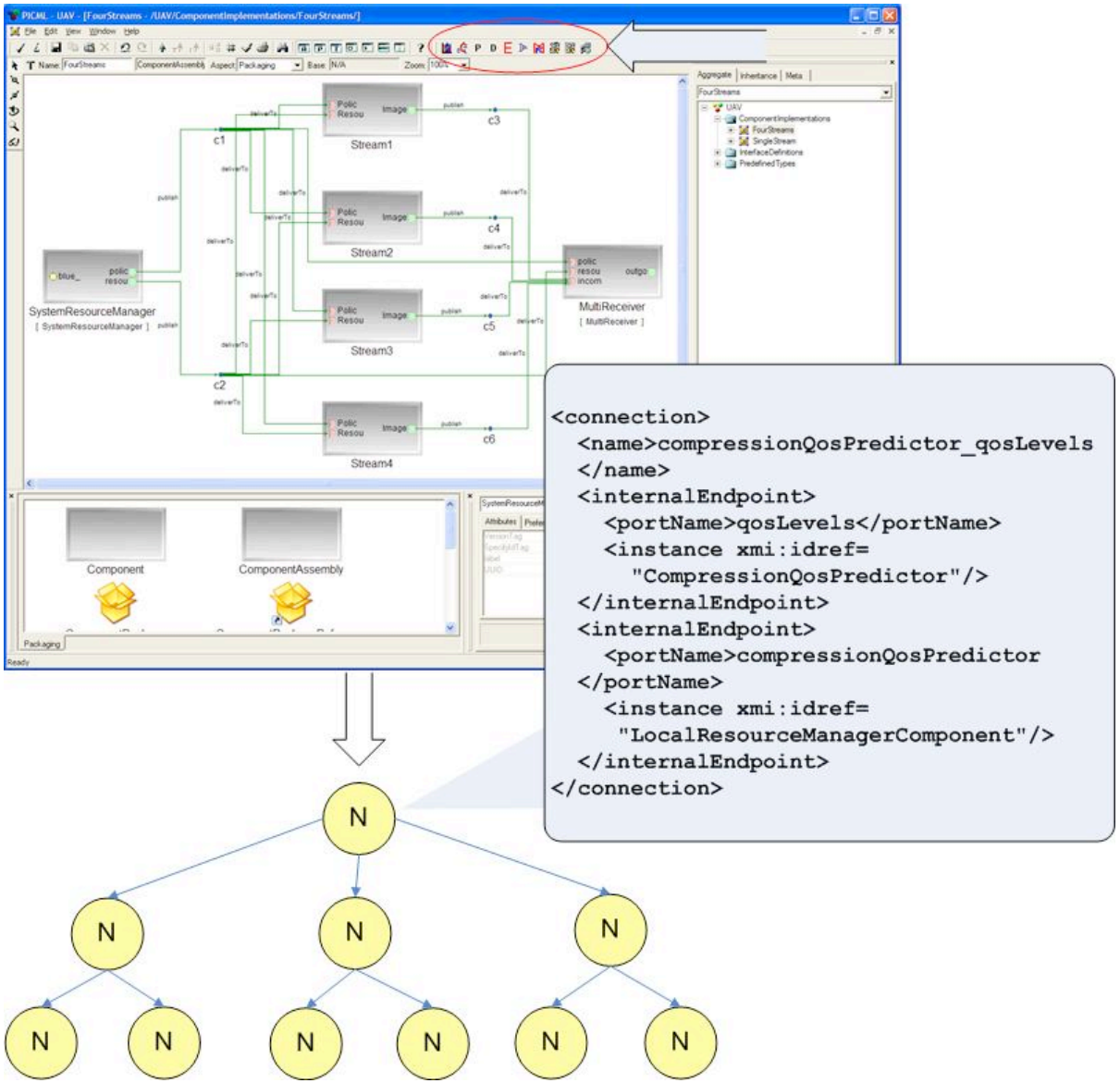
MDE Tool Development in GME

- Tool developers use MetaGME to develop a *domain-specific graphical modeling environment*
  - Define syntax & visualization of the environment via *metamodeling*
  - Define static semantics via *Object Constraint Language (OCL)*



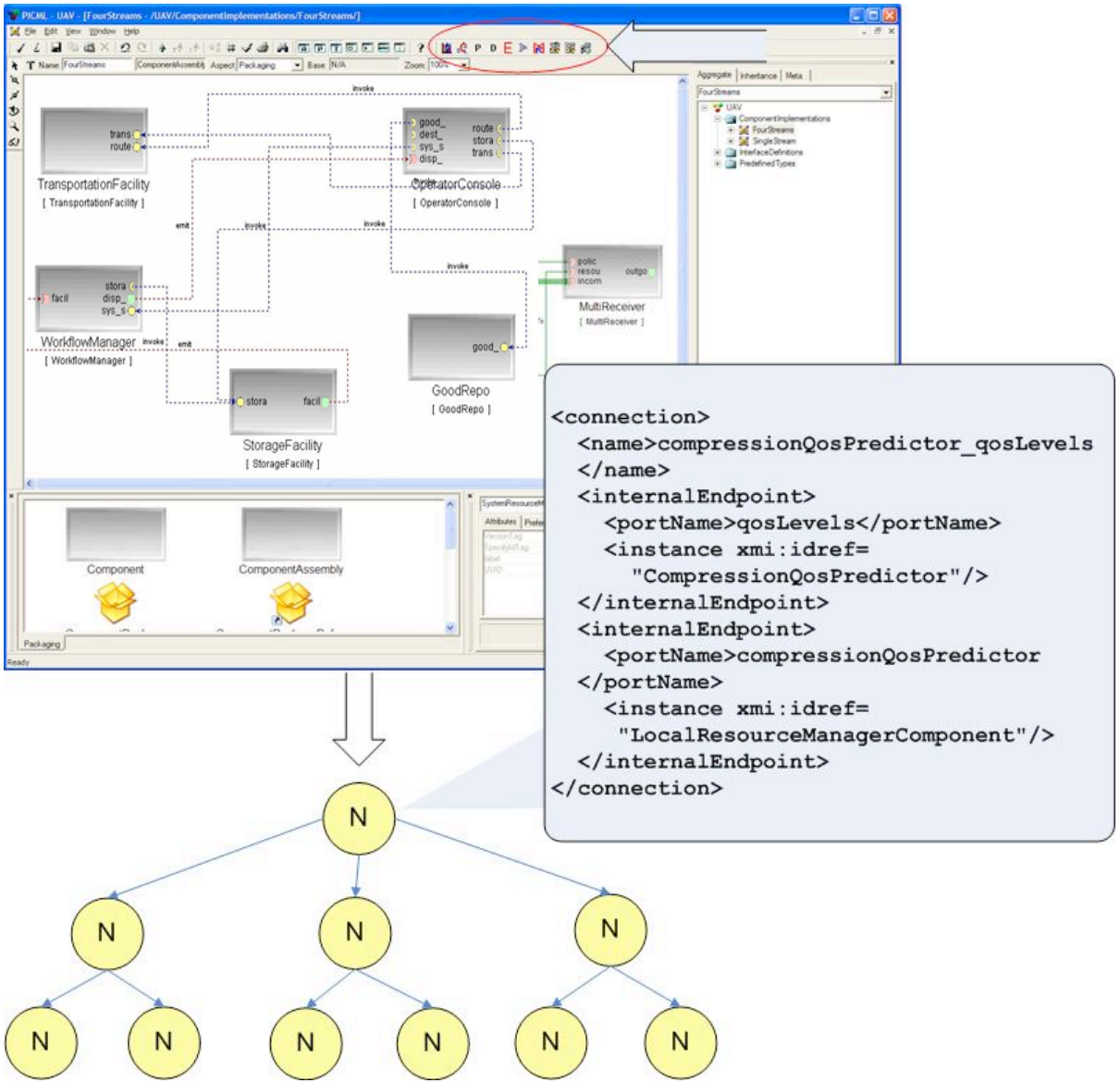
MDE Tool Development in GME

- Tool developers use MetaGME to develop a *domain-specific graphical modeling environment*
  - Define syntax & visualization of the environment via *metamodeling*
  - Define static semantics via *Object Constraint Language (OCL)*
  - Dynamic semantics implemented via *model interpreters*



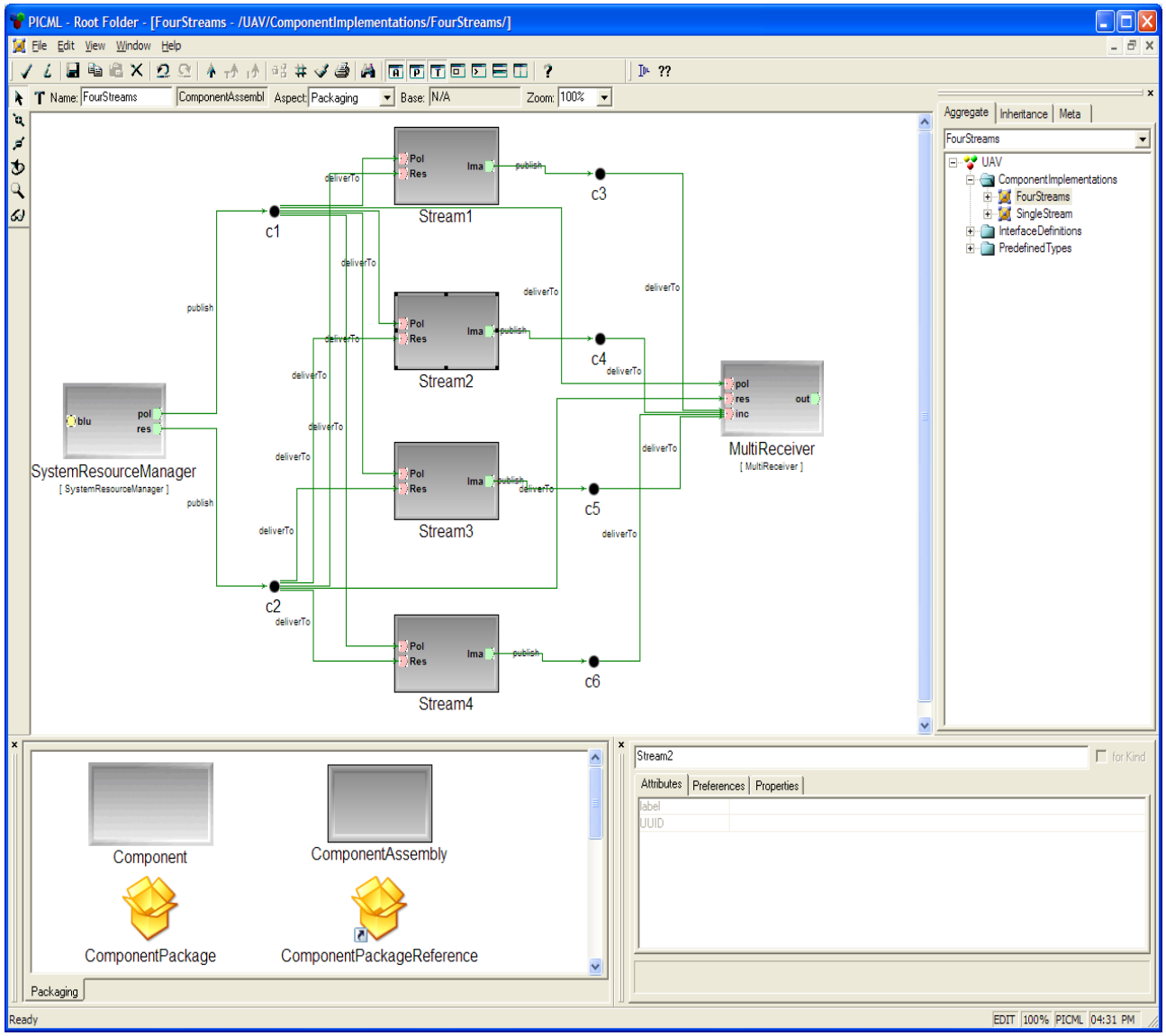
MDE Tool Development in GME

- Tool developers use MetaGME to develop a *domain-specific graphical modeling environment*
  - Define syntax & visualization of the environment via *metamodeling*
  - Define static semantics via *Object Constraint Language (OCL)*
  - Dynamic semantics implemented via *model interpreters*



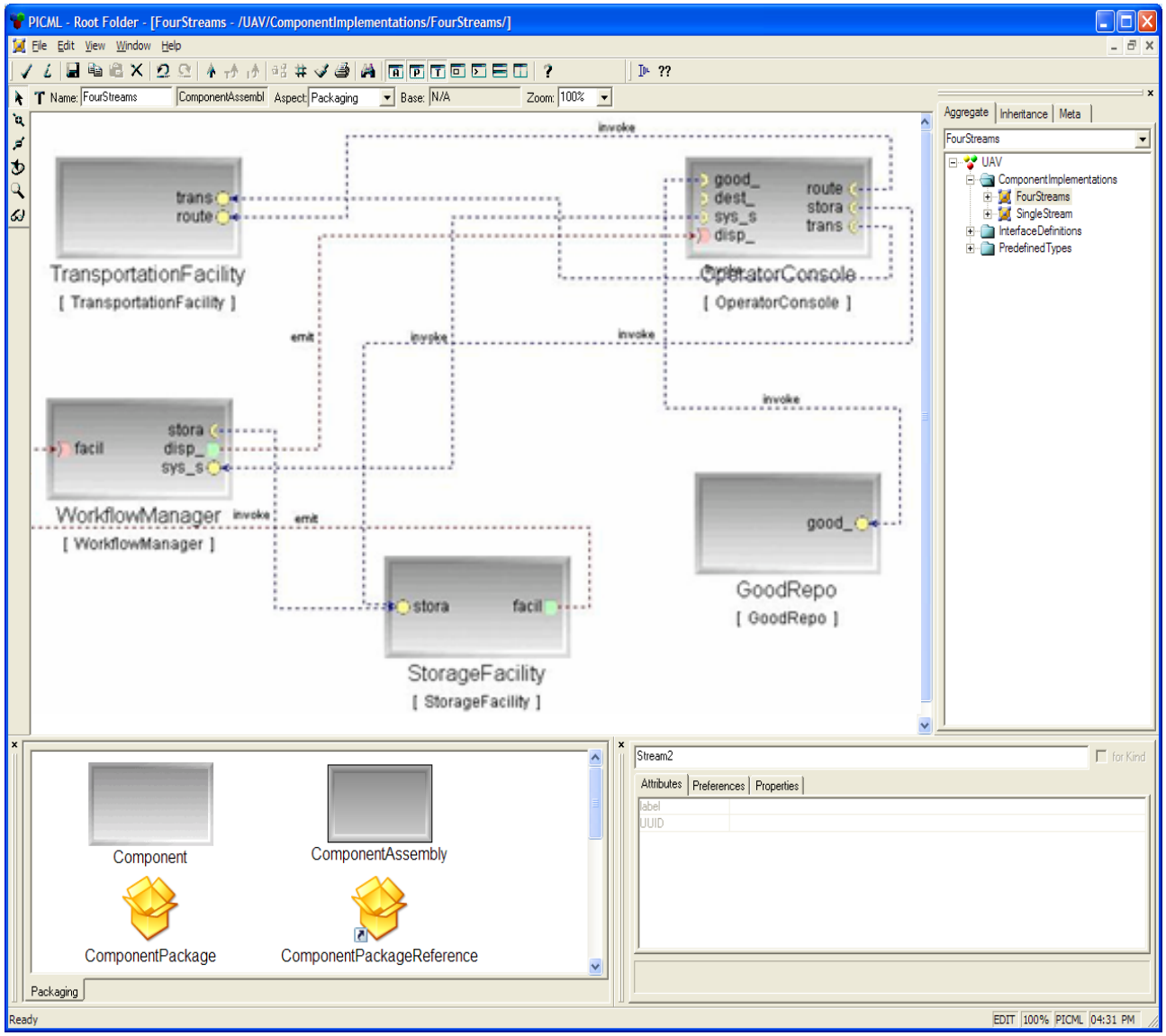
MDE Application Development with GME

- Application developers use modeling environments created w/MetaGME to build *applications*
  - Capture elements & dependencies visually



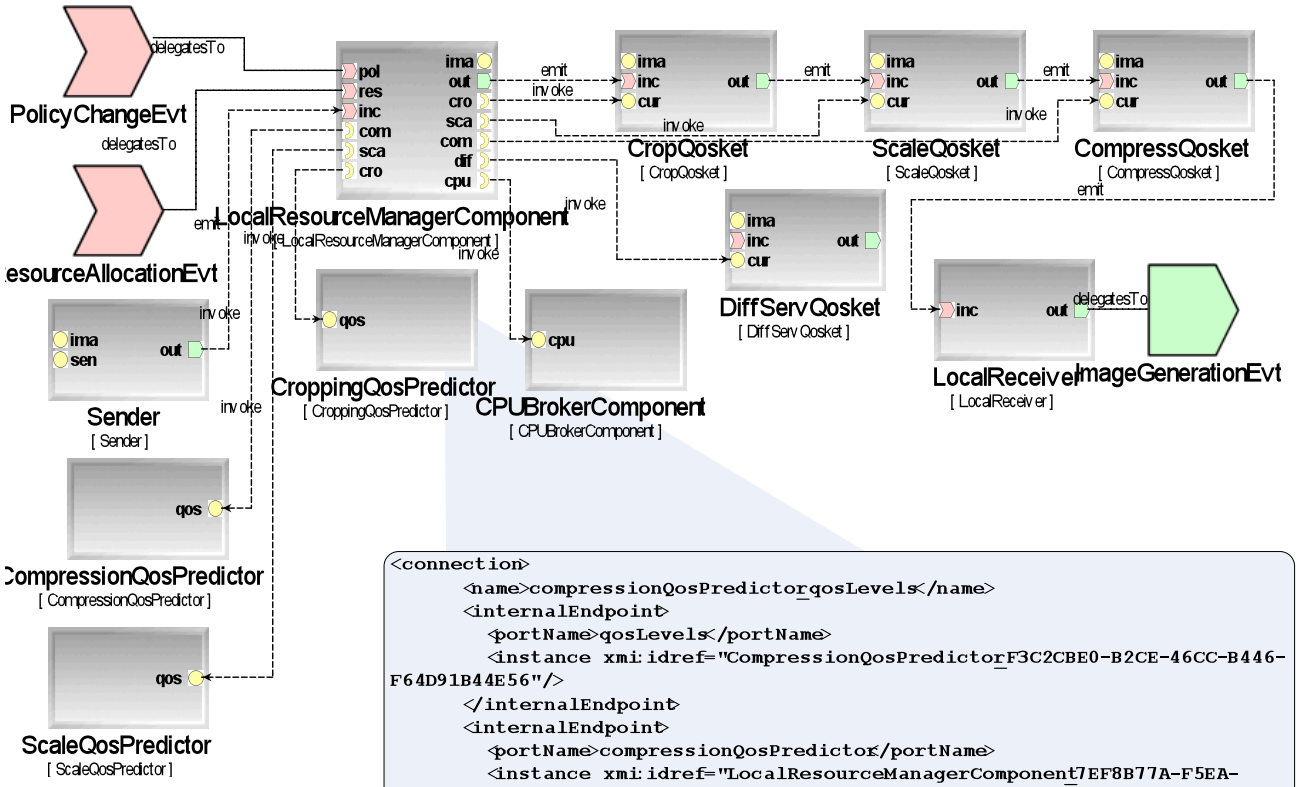
MDE Application Development with GME

- Application developers use modeling environments created w/MetaGME to build *applications*
  - Capture elements & dependencies visually



MDE Application Development with GME

- Application developers use modeling environments created w/MetaGME to build *applications*
  - Capture elements & dependencies visually
  - Model interpreter produces something useful from the models
    - e.g., code, simulations, deployment descriptions & configurations




```
<connection>
      <name>compressionQosPredictor_qosLevels</name>
      <internalEndpoint>
        <portName>qosLevels</portName>
        <instance xmi:idref="CompressionQosPredictor_F3C2CBE0-B2CE-46CC-B446-
F64D91B44E56"/>
      </internalEndpoint>
      <internalEndpoint>
        <portName>compressionQosPredictor</portName>
        <instance xmi:idref="LocalResourceManagerComponent_7EF8B77A-F5EA-
4D1A-942E-13AE7CFED30A"/>
      </internalEndpoint>
    </connection>
    <connection>
      <name>scalingQosPredictor_qosLevels</name>
      <internalEndpoint>
        <portName>qosLevels</portName>
        <instance xmi:idref="ScaleQosPredictor_F3024A4F-F6E8-4B9A-BD56-
A2E802C33E32"/>
      </internalEndpoint>
      <internalEndpoint>
        <portName>scalingQosPredictor</portName>
        <instance xmi:idref="LocalResourceManagerComponent_7EF8B77A-F5EA-
4D1A-942E-13AE7CFED30A"/>
      </internalEndpoint>
    </connection>
```

Applying MDE to Component-based Development

Specification & Implementation

- Defining, partitioning, & implementing application functionality as standalone components

Assembly & Packaging

- Bundling a suite of software binary modules & metadata representing app components

Installation

- Populating a repository with packages required by app

Configuration

- Configuring packages with appropriate parameters to satisfy functional & systemic requirements of an application without constraining to physical resources

Planning

- Making deployment decisions to identify nodes in target environment where packages will be deployed

Preparation

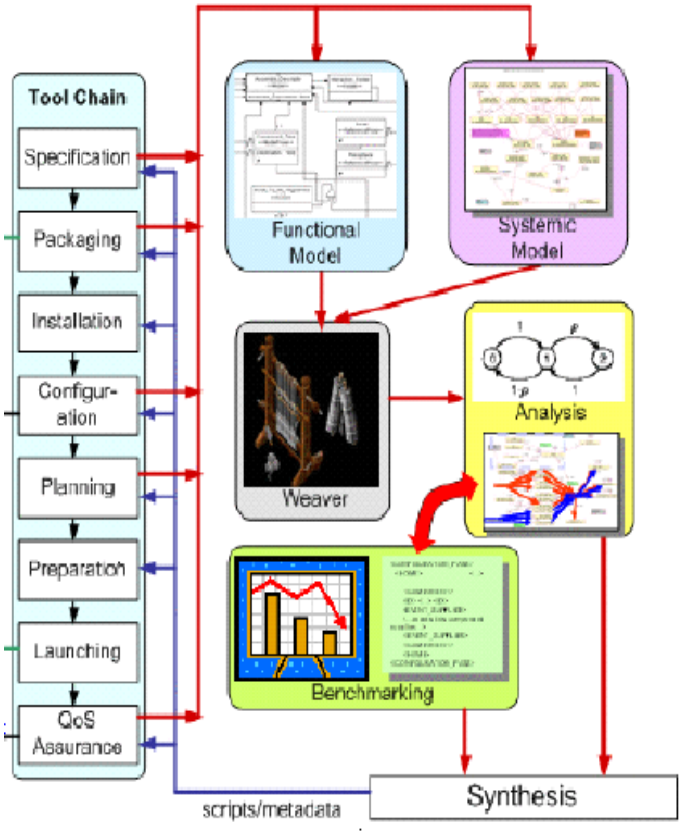
- Moving binaries to identified entities of target environment

Launching

- Triggering installed binaries & bringing appln to ready state

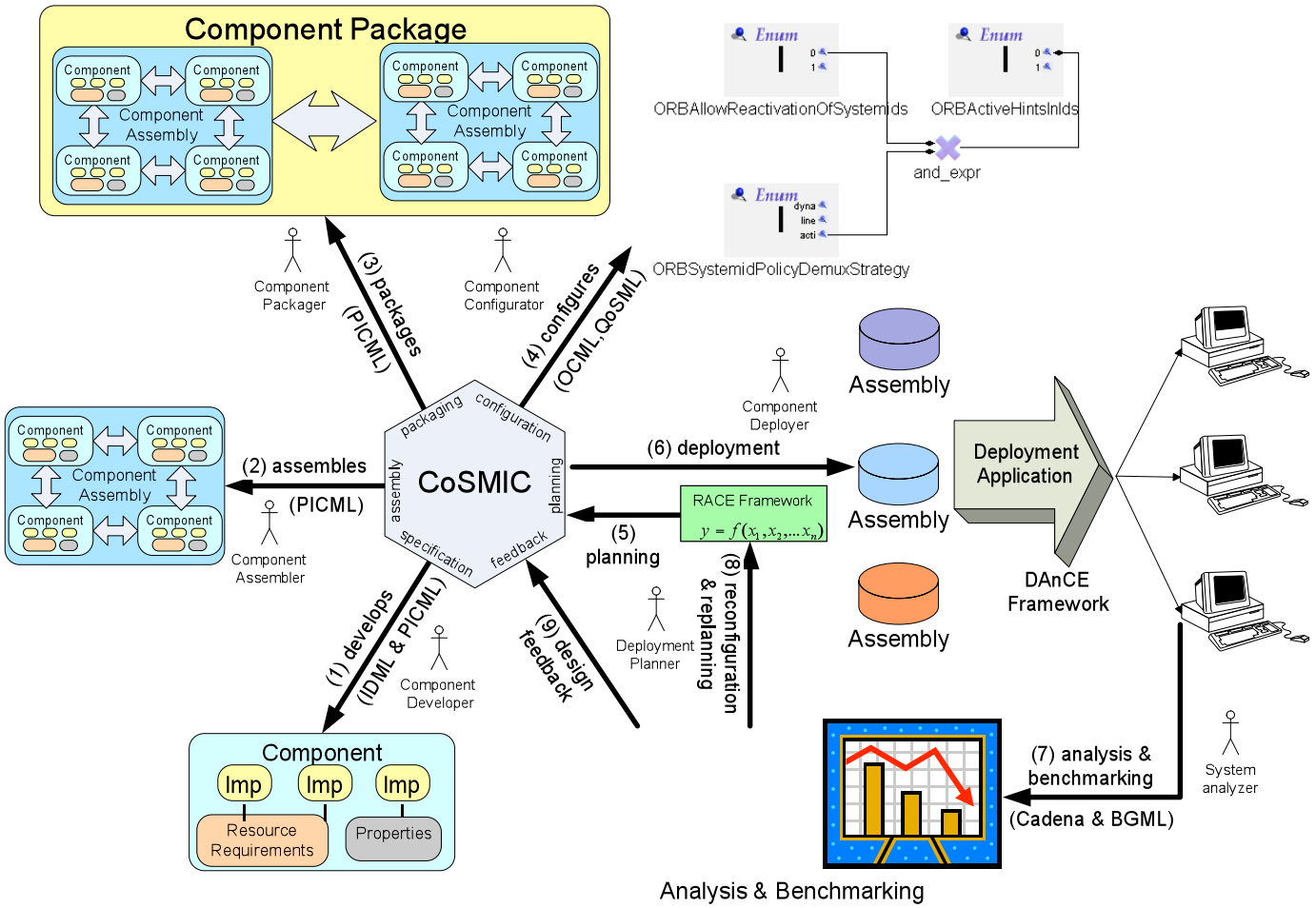
QoS Assurance & Adaptation

- QoS validation, runtime (re)configuration & resource management to maintain end-to-end QoS



OMG Deployment & Configuration (D&C) specification (ptc/05-01-07)

Our MDE Solution: CoSMIC Tool Suite

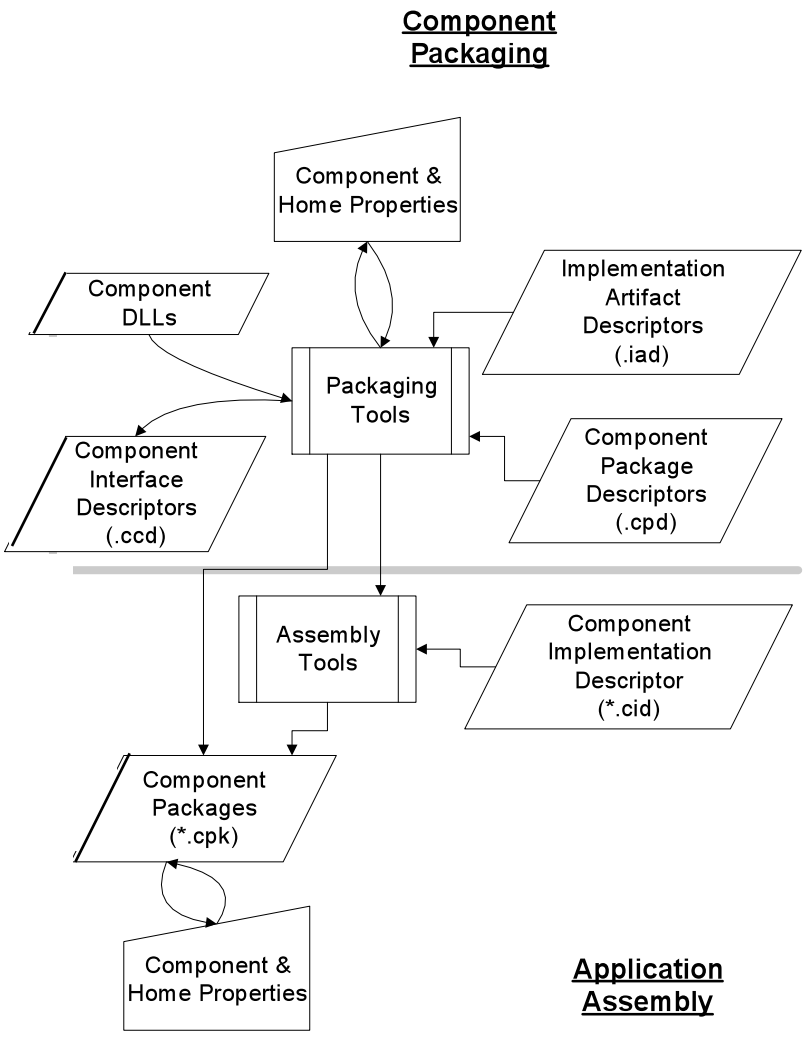


- CoSMIC tools, *e.g.*, PICML used to model application components
- Captures the data model of the OMG D&C specification
- Synthesis of static deployment plans for DRE applications

CoSMIC can be downloaded at www.dre.vanderbilt.edu/cosmic

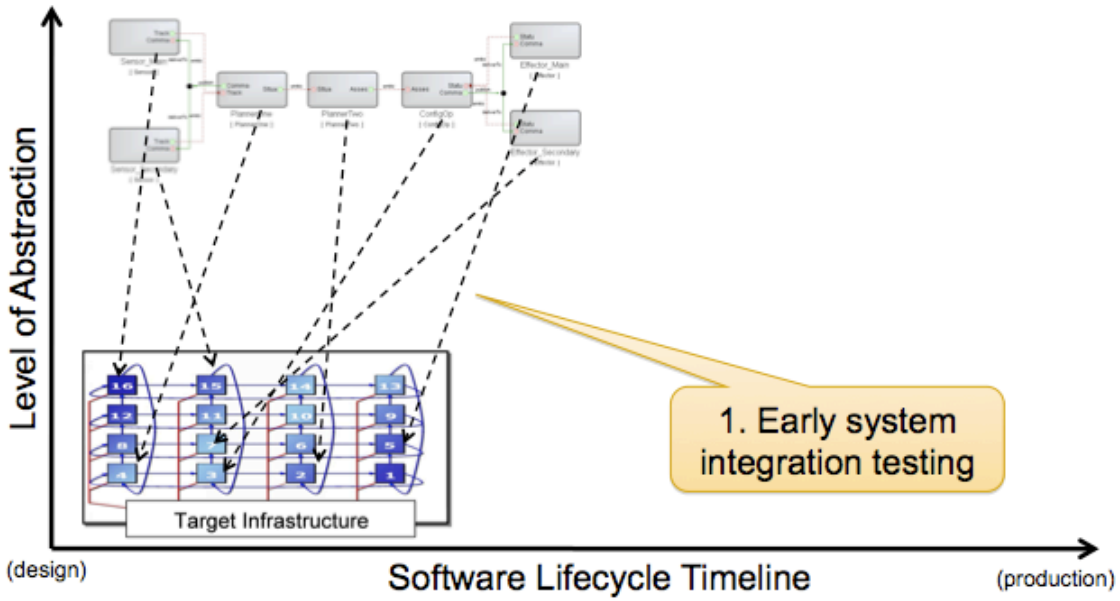
Example Metadata Generated by PICML

- **Component Interface Descriptor (.ccd)**
  - Describes the interface, ports, properties of a single component
- **Implementation Artifact Descriptor (.iad)**
  - Describes the implementation artifacts (e.g., DLLs, OS, etc.) of one component
- **Component Package Descriptor (.cpd)**
  - Describes multiple alternative implementations of a single component
- **Package Configuration Descriptor (.pcd)**
  - Describes a configuration of a component package
- **Top-level Package Descriptor (package.tpd)**
  - Describes the top-level component package in a package (.cpk)
- **Component Implementation Descriptor (.cid)**
  - Describes a specific implementation of a component interface
  - Implementation can be either monolithic- or assembly-based
  - Contains sub-component instantiations in case of assembly based implementations
  - Contains inter-connection information between components
- **Component Packages (.cpk)**



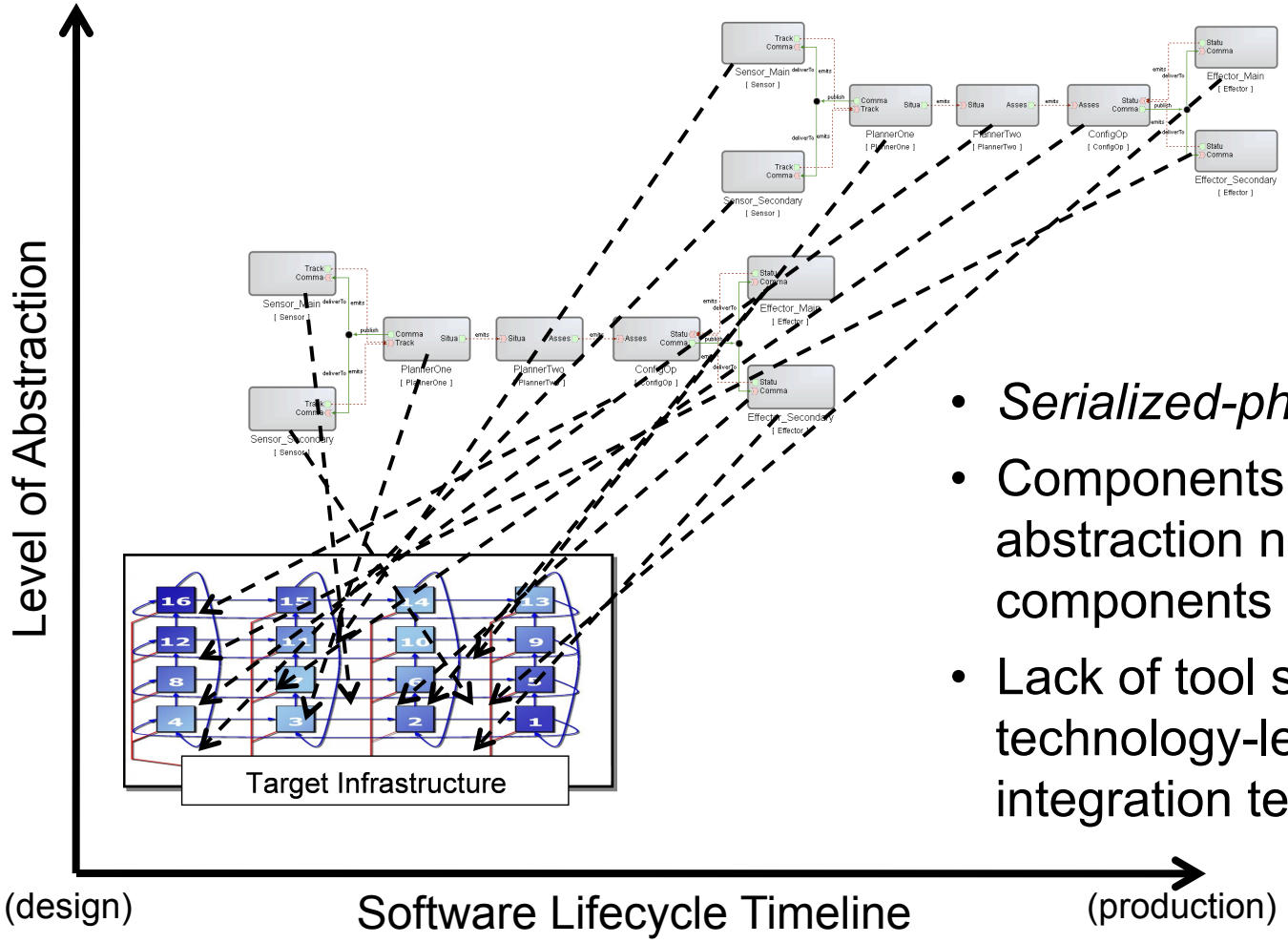
Based on OMG (D&C) specification (ptc/03-06-03)

PART 3:
Early System Integration Testing – Addressing Complexity #1



Early System Integration Testing

Conventional component-based systems have some characteristics that prevent early system integration testing, such as:



- *Serialized-phasing* development
- Components within same level of abstraction need other components
- Lack of tool support at technology-level for early system integration testing

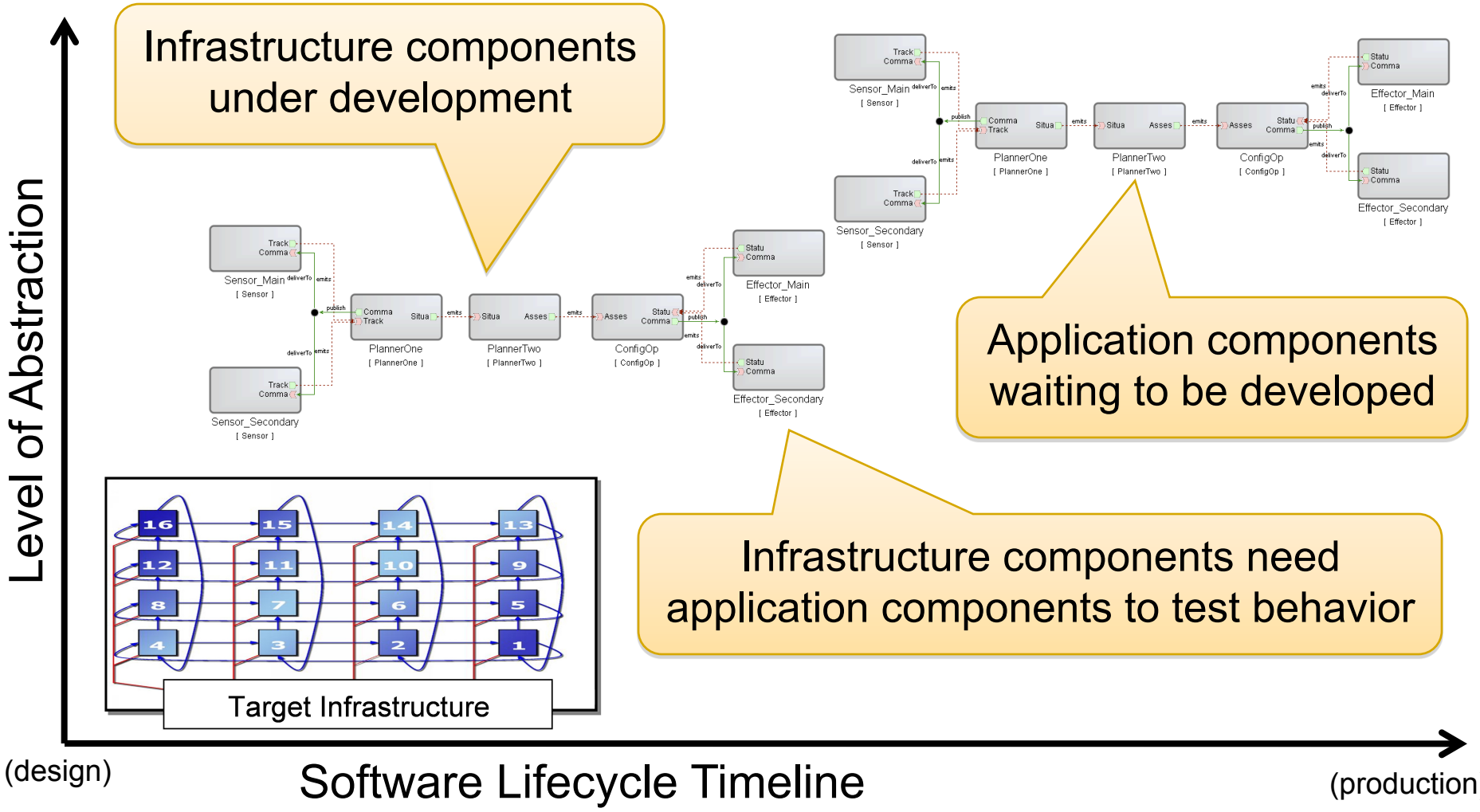
Early System Integration Testing Properties

How to facilitate early integration testing of DRE component-based systems for software developers & testers?

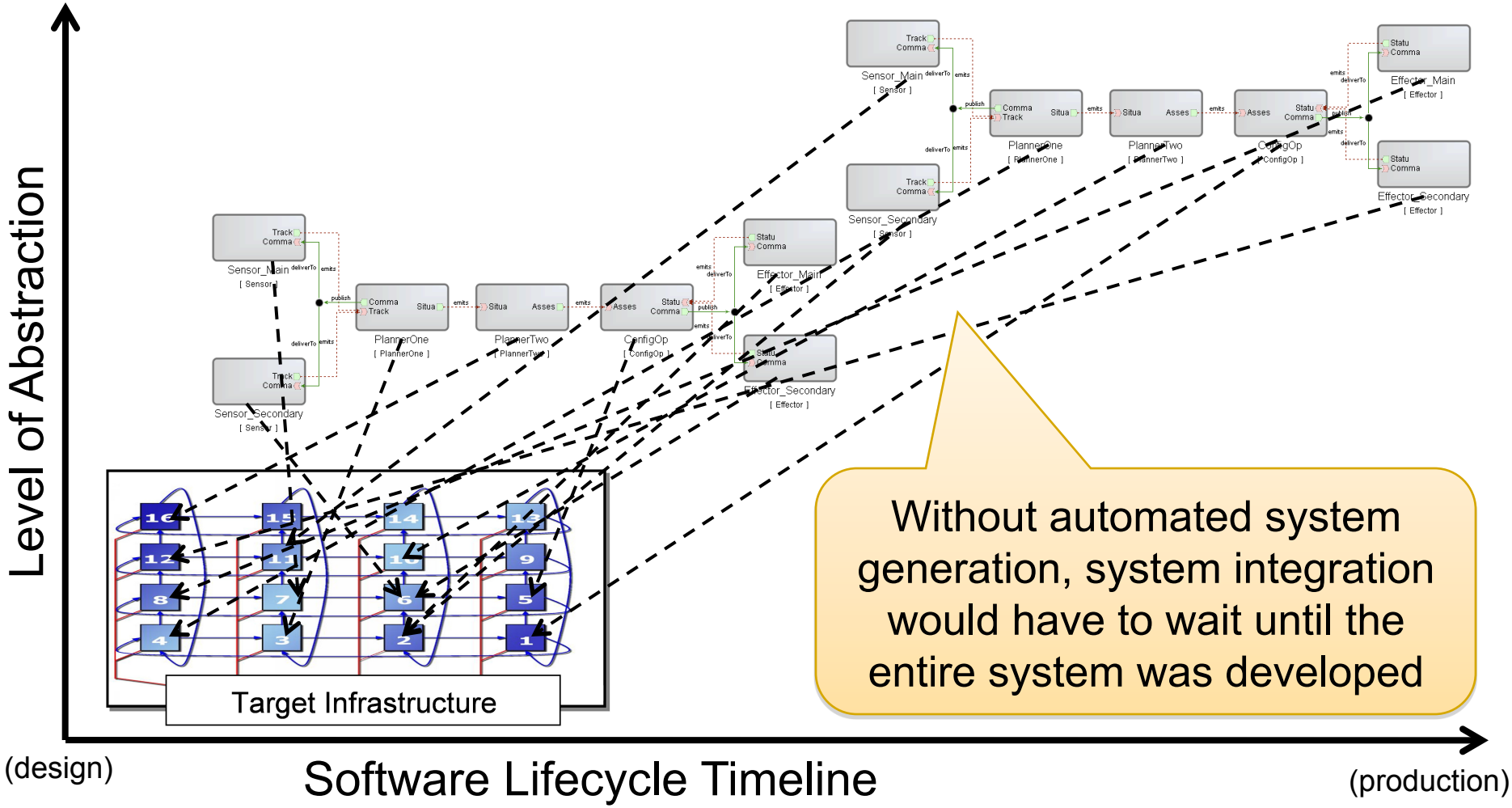
Property	Description
Automation	Technique leverages automation to generate a system for early testing
Target Architecture	Technique allows developers & testers to do early testing on their target architecture
Granularity	Developers can model system behavior at different levels of granularity
Replaceable	As developers implement the real components, they can easily replace the faux components

Note: Evaluation criteria is based on characteristics of enterprise component-based systems & enterprises trying to apply agile development techniques on such systems

Challenge: Test System Auto-generation (1/3)



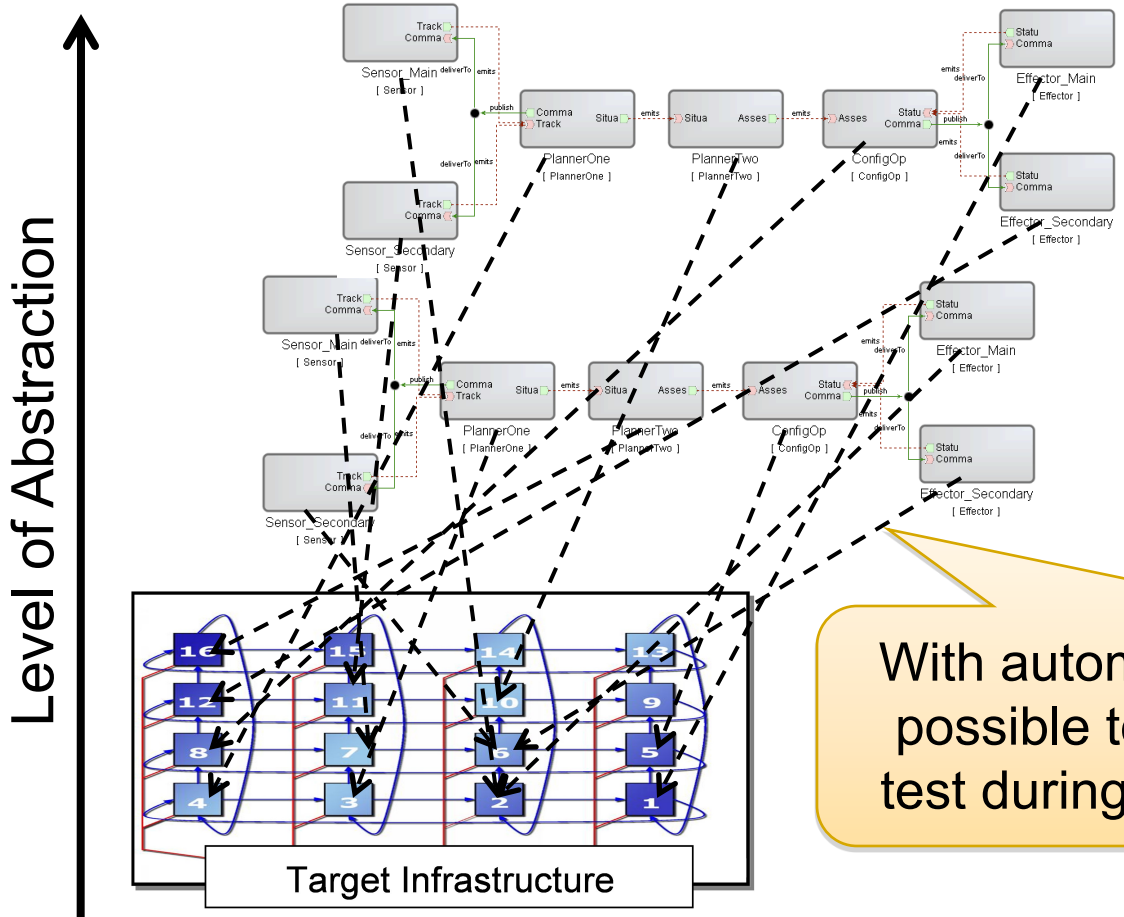
Challenge: Test System Auto-generation (2/3)



Serialized Phasing

- Performance/integration testing can only begin once infrastructure & application components are completely developed

Challenge: Test System Auto-generation (3/3)



Benefits of Auto-generation

- Reduces complexity of developing test applications & systems in parallel with main development efforts
- Shift focus to “testing-logic” instead of test implementation

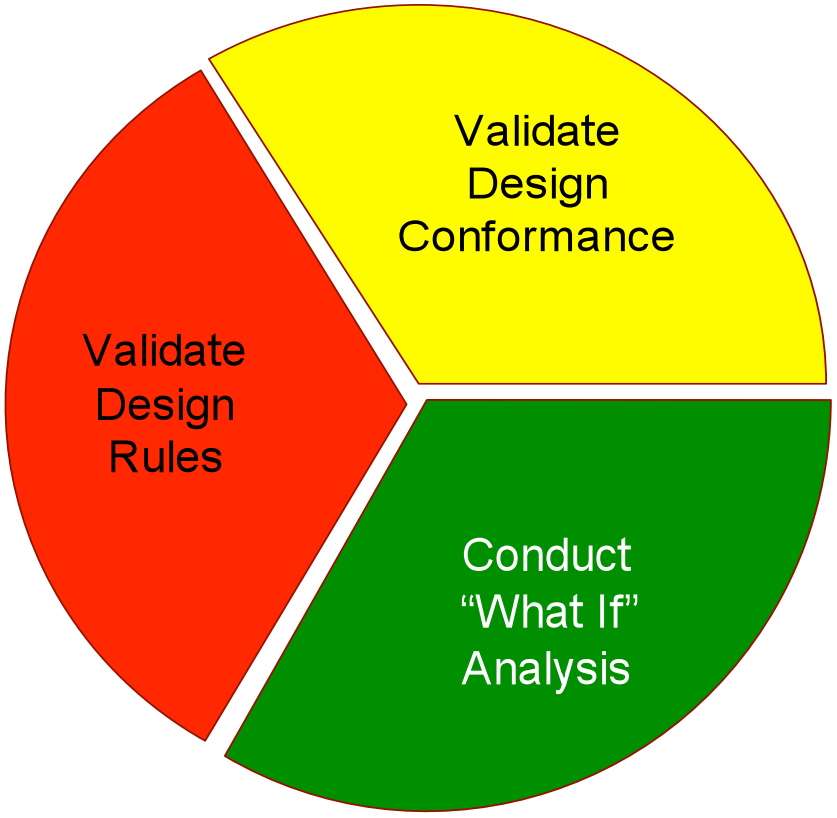
With automated system generation, it is possible to perform system integration test during early stages of development

Solution: SEM Execution Modeling Tools

Component Workload Emulator (CoWorkEr) Utilization Test Suite (CUTS) Workflow

While target system under development:

1. Use a *domain-specific modeling language* (DSML) to define & validate infrastructure specifications & requirements
   - *e.g.*, the Platform Independent Component Modeling Language (PICML)
2. Use DSML to define & validate application specifications & requirements
3. Use middleware & MDD tools generate D&C metadata so system conforms to its specifications & requirements
4. Use analysis tools to evaluate & verify QoS performance
5. Redefine system D&C & repeat

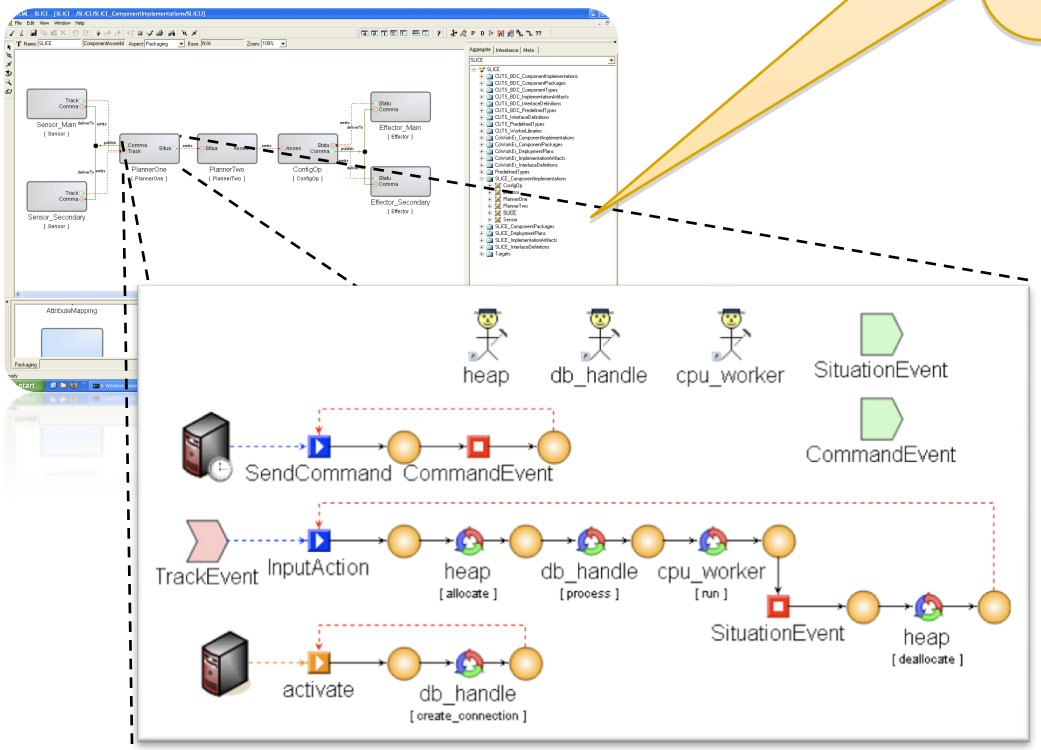


Enables early testing & feedback on target infrastructure throughout the development lifecycle

CUTS Workflow & Architecture (1/5)

- System execution modeling (SEM) tools are a promising technology for addressing serialized-phasing problems

Without domain-specific modeling languages (DSMLs), developers must work outside their knowledge domain to model system behavior



Component Behavior Modeling Language (CBML)

- High-level behavior DSML based on Timed Input/Output Automata (TIOA)

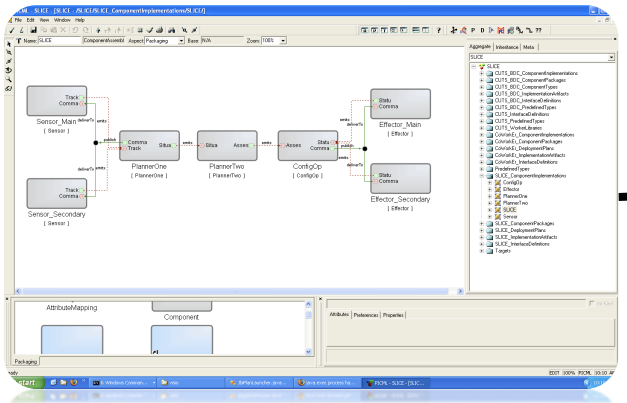
Workload Modeling Language (WML)

- Parameterizes generic CBML actions with well-defined workloads

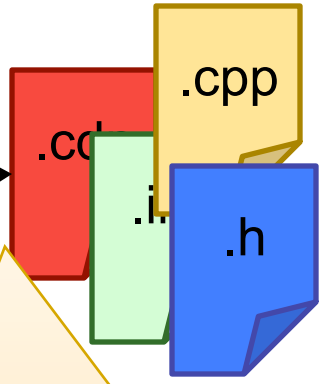
CBML & WML are platform-, language-, & technology-independent DSMLs designed to integrate with structural DSMLs

CUTS Workflow & Architecture (2/5)

- System execution modeling (SEM) tools are a promising technology for addressing serialized-phasing problems



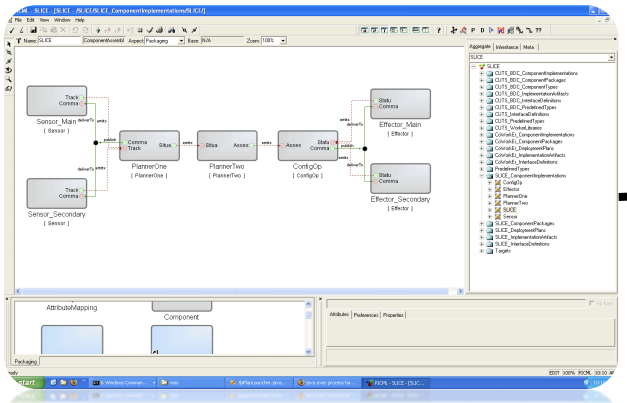
generates



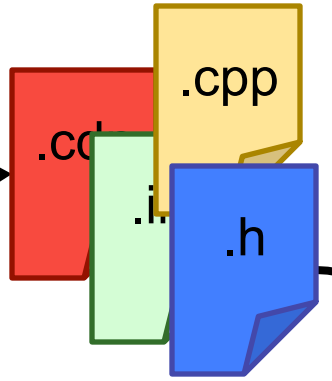
Developers **do not** have to spend substantial time/effort manually implementing test applications **in parallel** with implementing real applications

CUTS Workflow & Architecture (3/5)

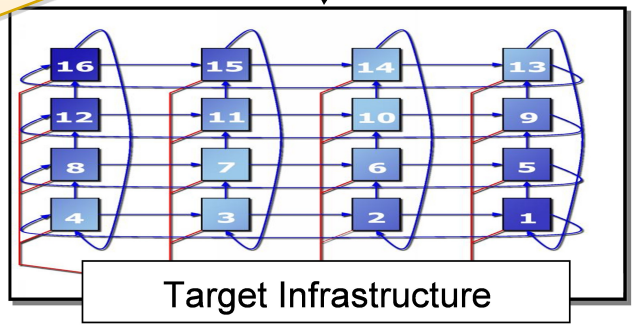
- System execution modeling (SEM) tools are a promising technology for addressing serialized-phasing problems



generates



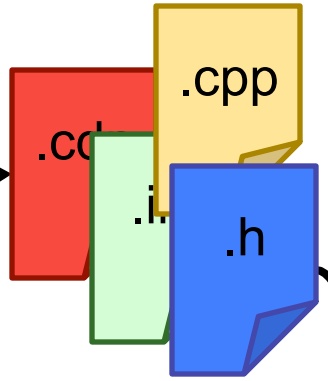
executes on



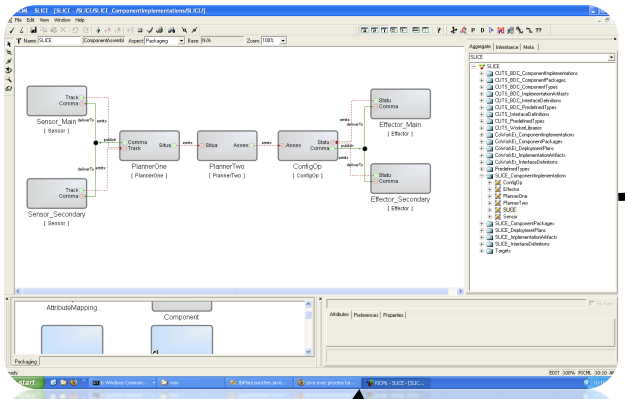
Developers **do not** have to wait until **complete system integration** to begin systemic performance testing

CUTS Workflow & Architecture (4/5)

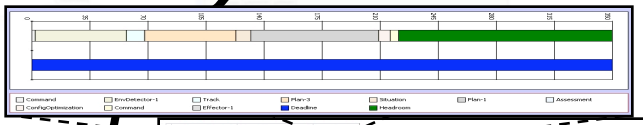
- System execution modeling (SEM) tools are a promising technology for addressing serialized-phasing problems



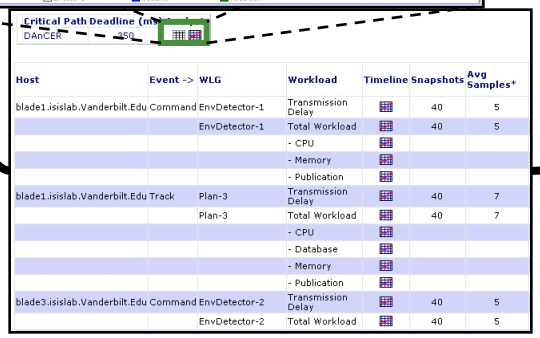
generates



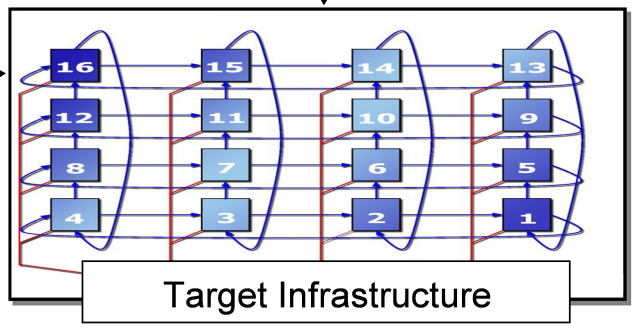
Developers **do not** have to spend substantial time/effort developing a framework for visualizing performance results



feedback

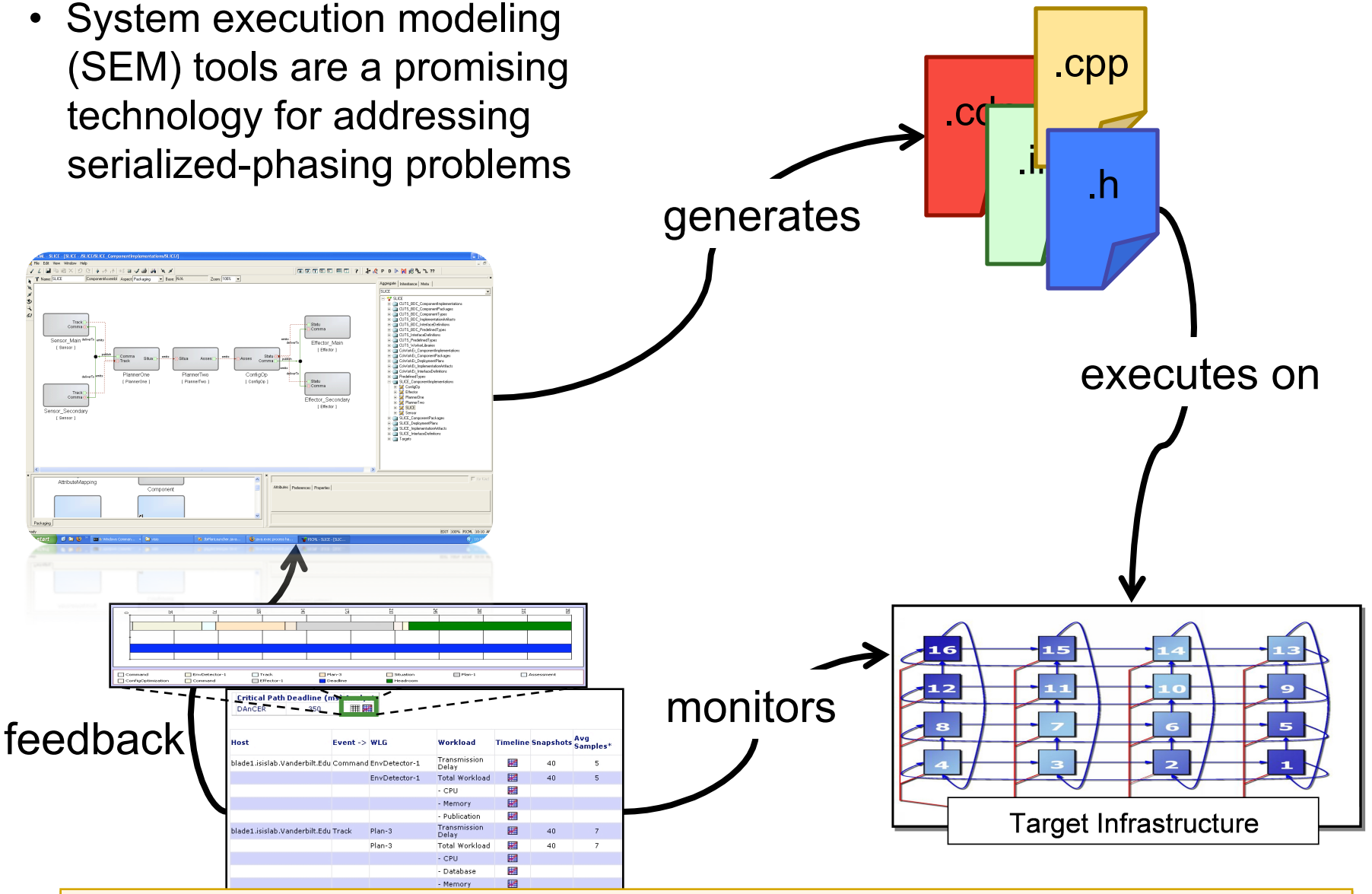


monitors



CUTS Workflow & Architecture (5/5)

- System execution modeling (SEM) tools are a promising technology for addressing serialized-phasing problems

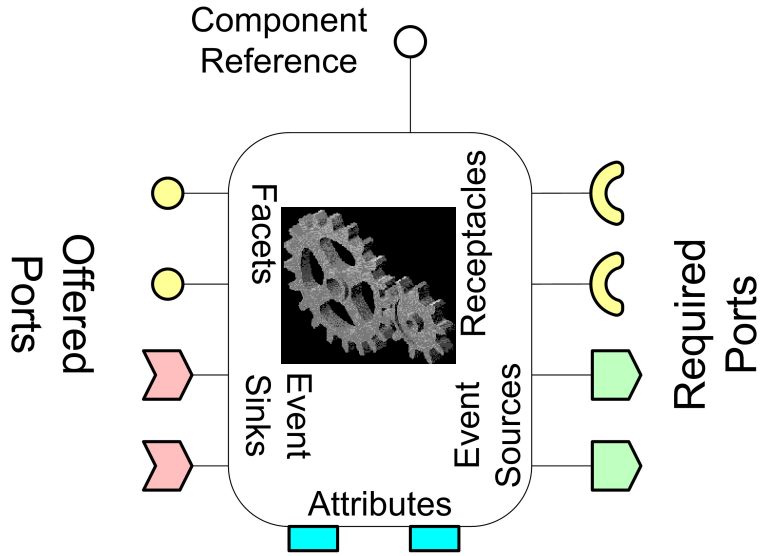


Enables *early* system integration testing on target infrastructure

The Component Behavior Modeling Language

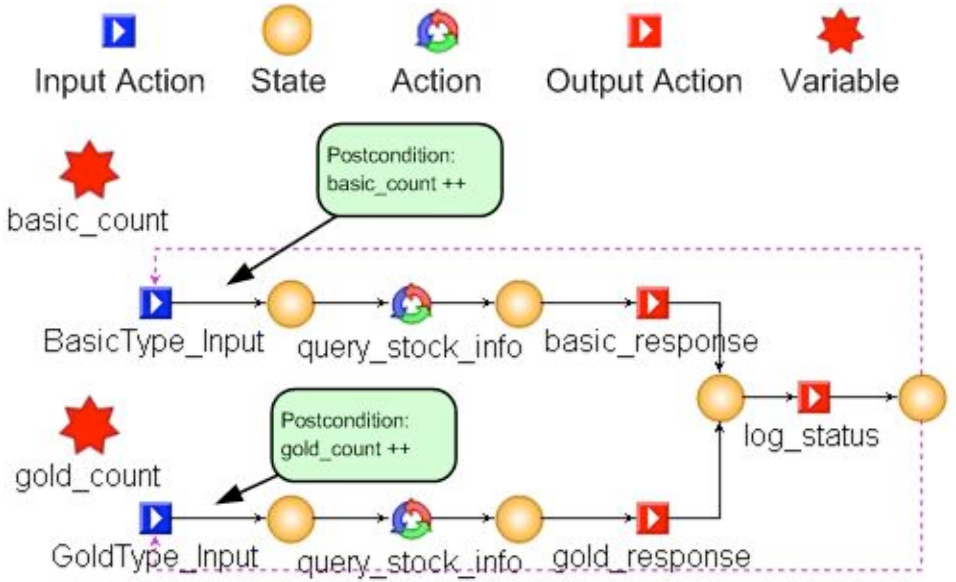
Context

- Component's behavior can be classified as: *communication* & *internal* actions
- Need to define these actions as close as possible to their real counterpart



Overview of CBML

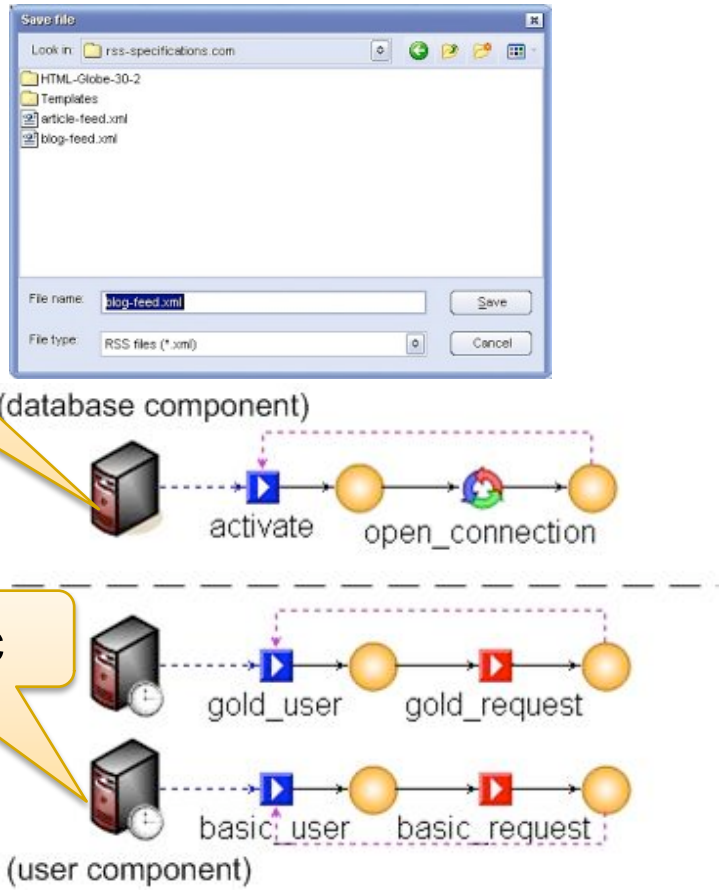
- Based on the semantics of Input /Output (I/O) Automata
- Behavior is specified using a series of *action* to *state* transitions
- *Transitions* have *preconditions* that represent guards & *effects* have *postconditions* that determine the new state after an action occurs
- *Variables* are used to store state & can be used within pre & postconditions



Domain-Specific Extensions to CBML

Context

- Some aspects of component-based systems are not first-class entities in I/O automata
  - e.g., lifecycle events & monitoring notification events
- Extended CBML (without affecting formal semantics) to support domain-specific extensions



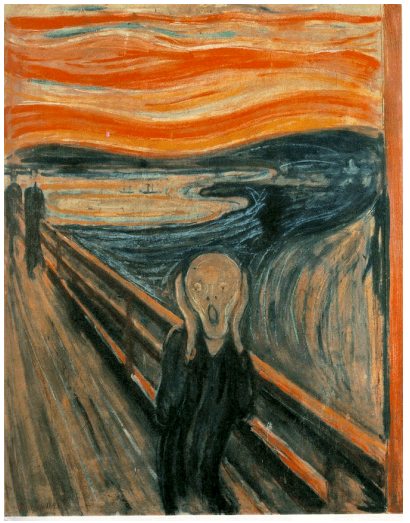
Domain-Specific Extensions

- *Environment events* – input actions to a component that are triggered by the hosting system rather than another component
- *Periodic events* - input actions from the hosting environment that occur periodically

Ensuring Scalability of CBML

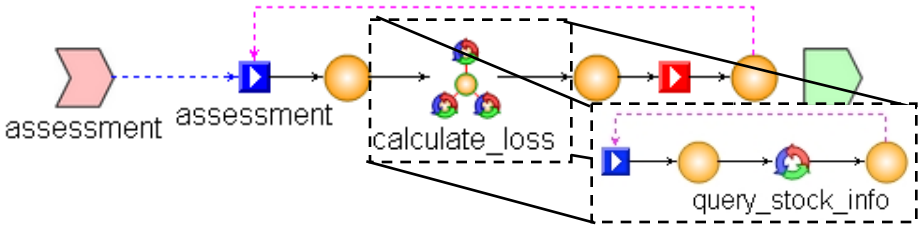
Context

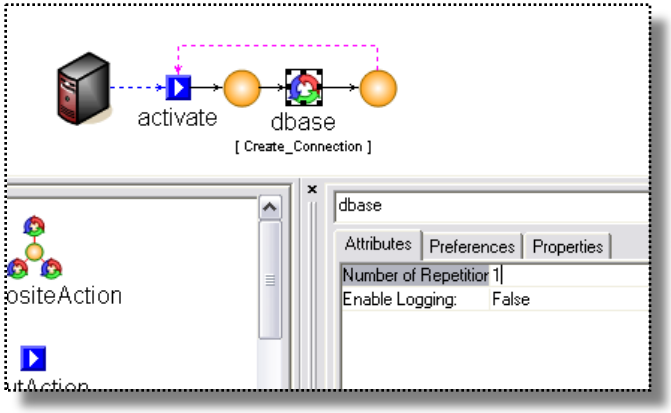
- One of the main goals of higher level of abstraction is *simplicity* & *ease of use*
- It is known that one of the main drawbacks of automata languages is *scalability*



Usability Extensions

- *Composite Action* – contains other actions & helps reduce model clutter
- *Repetitions* - determines how many times to repeat an action to prevent duplicate sequential actions
- *Log Action* - an attribute of an Action element that determines if the action should be logged





Tool Specific – GME add-on that auto-generates required elements (e.g., states)

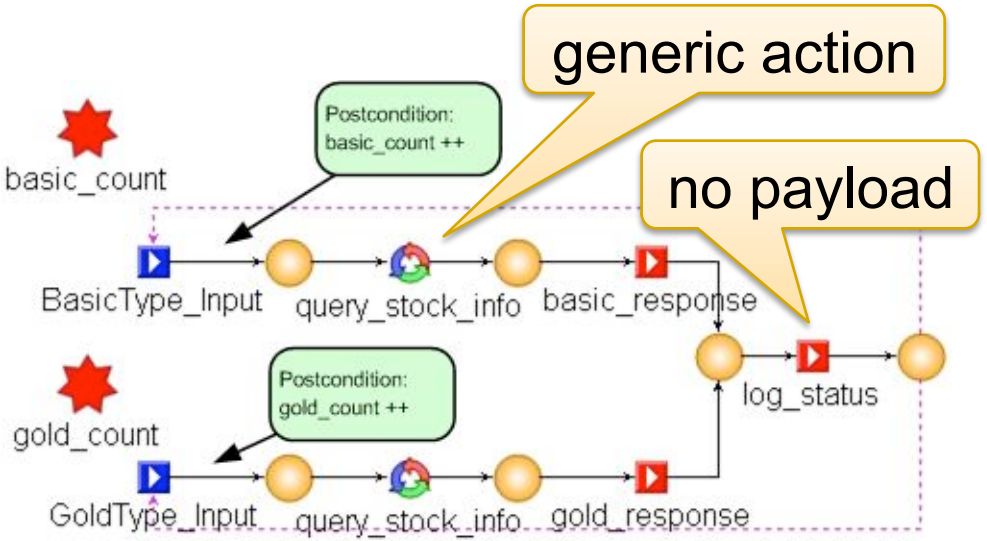
The Workload Modeling Language

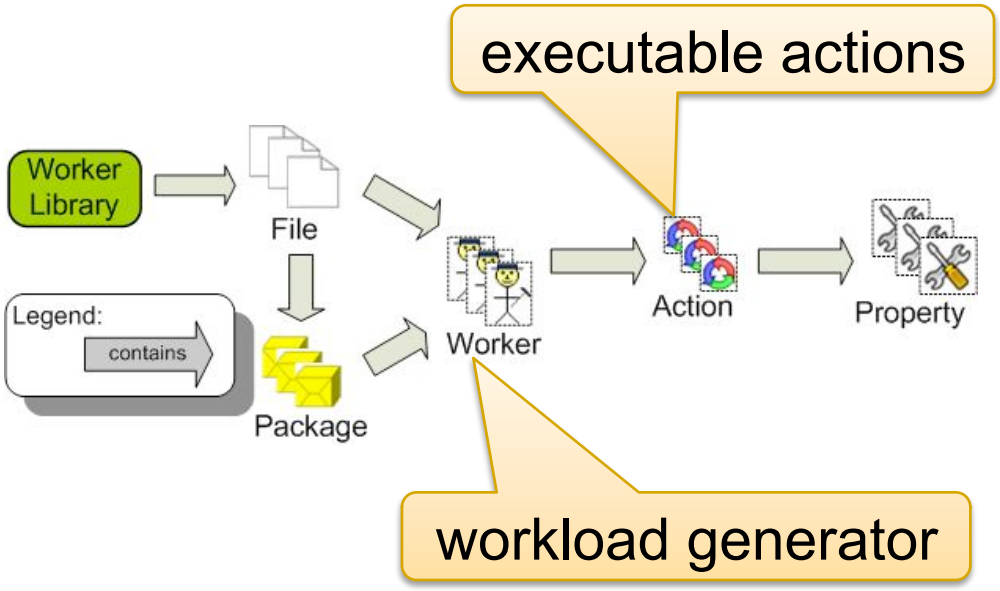
Context

- CBML as a standalone language is sufficient enough to capture behavior of a component
- For early integration testing, it does not capture the reusable objects of a component & its workload

Overview of WML

- Middleware, hardware, platform, & programming language independent DSML
- Used to define workload generators (workers) that contains actions to represent realistic operations
- Defined using a hierarchical structure that resembles common object-oriented programming packaging techniques
  - *i.e.*, consistent with conventional component technologies





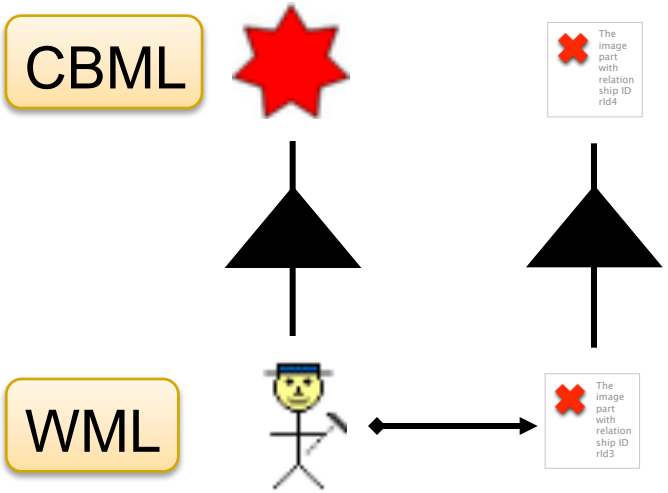
Integrating WML Models with CBML Models

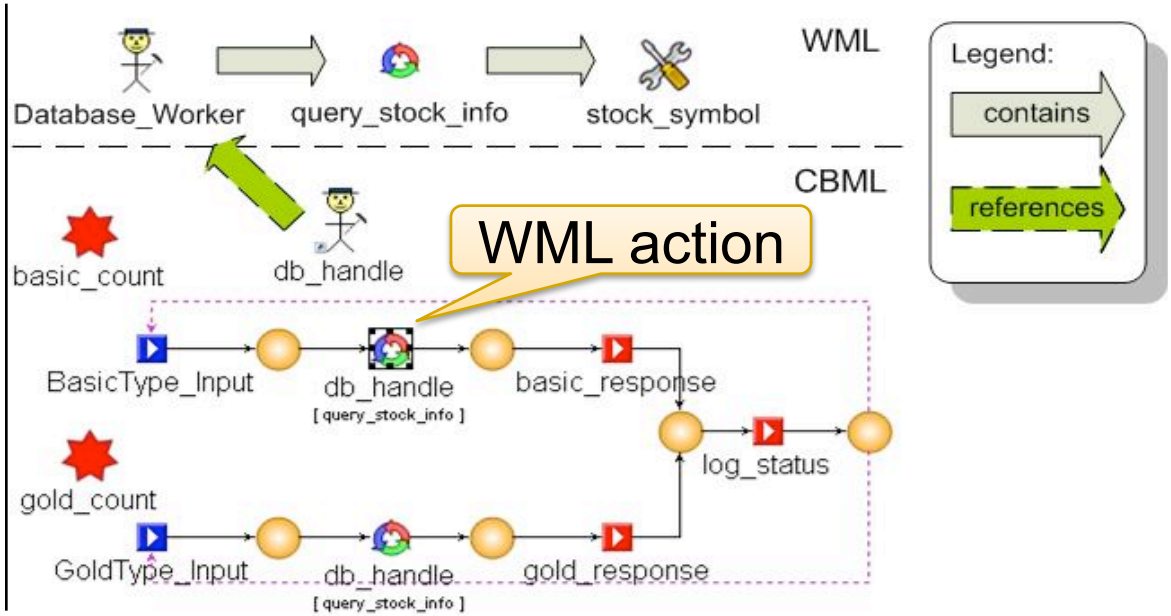
Context

- CBML & WML are standalone DSMLs with a distinct purpose
  - i.e., model behavior & workload, respectively
- WML is designed to complement CBML by providing CBML with reusable operations that can map to realistic operations

Integration Enablers

- WML *worker* elements have same model semantics as *variables*
- WML *actions* have same modeling semantics as CBML actions
- Allows WML elements to be used in CBML models

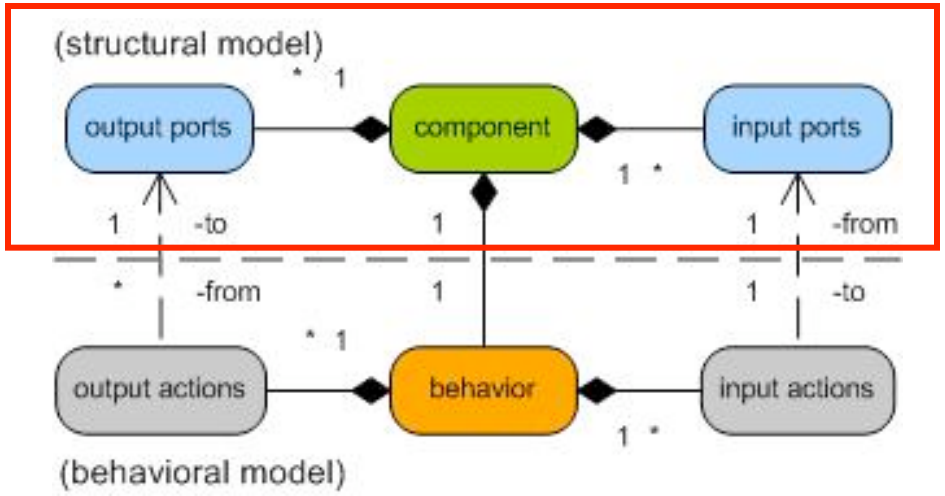




Integrating Behavioral and Structural DSMLs

Context

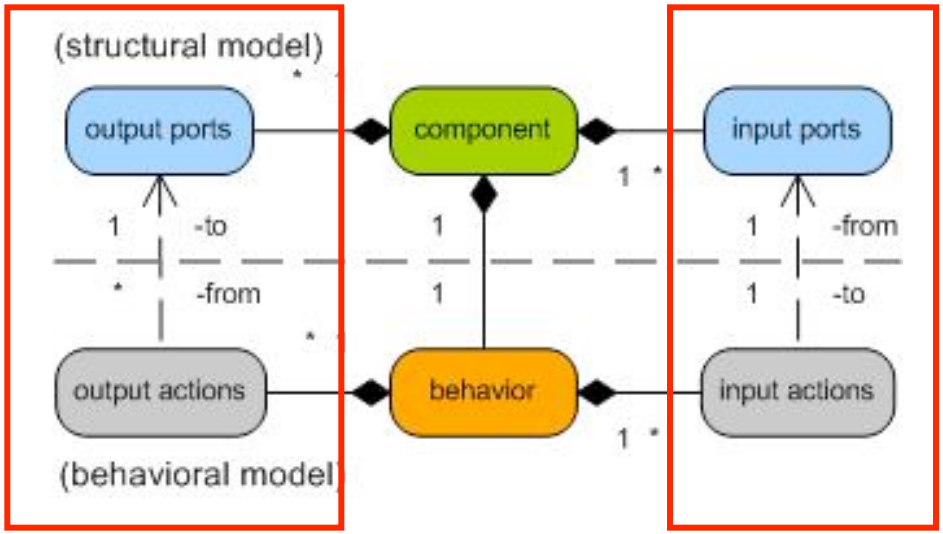
- Structural DSML (e.g., the Platform Independent Component Modeling Language (PICML)) capture the makeup of component-based systems



Integrating Behavioral and Structural DSMLs

Context

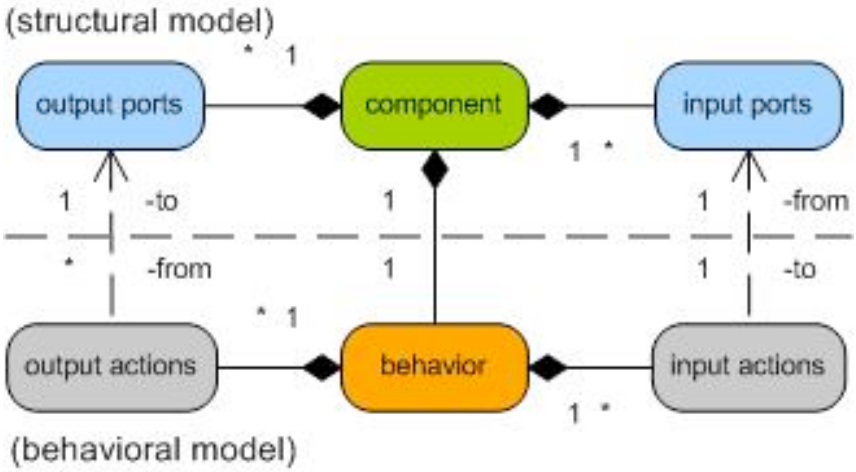
- Structural DSML (e.g., the Platform Independent Component Modeling Language (PICML)) capture the makeup of component-based systems
- There is no correlation between a component's ports & its behavior



Integrating Behavioral and Structural DSMLs

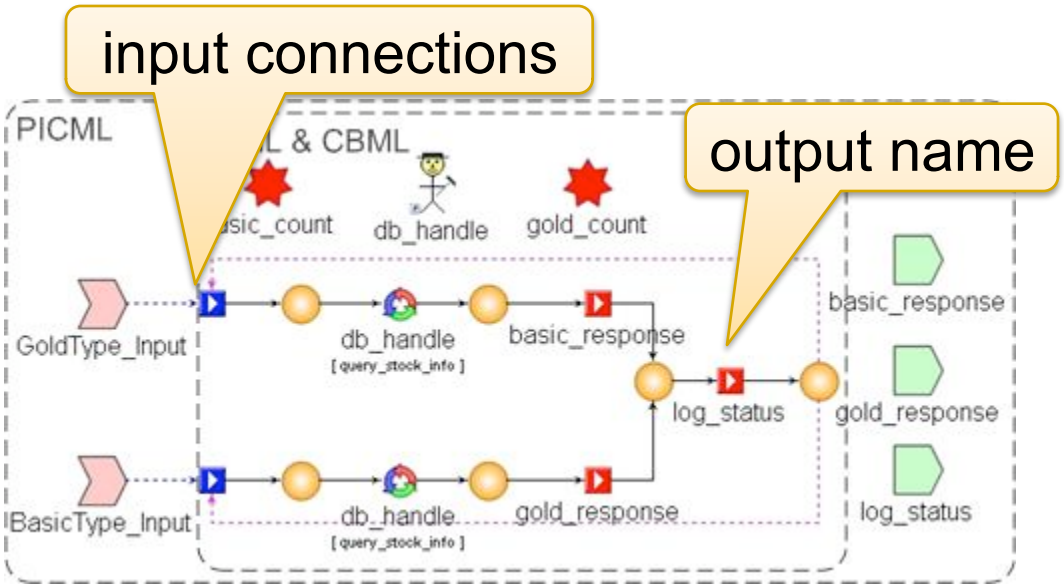
Context

- Structural DSML (e.g., the Platform Independent Component Modeling Language (PICML)) capture the makeup of component-based systems
- There is no correlation between a component's ports & its behavior



Integration Enablers

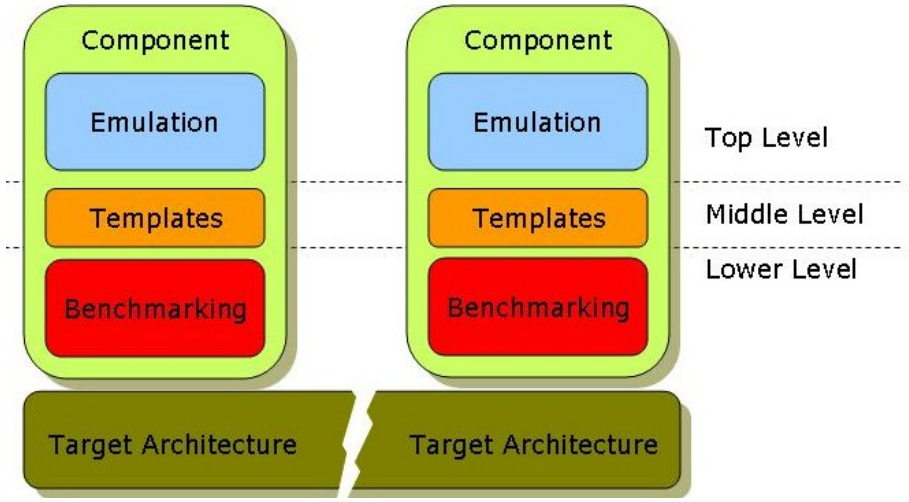
- Defined a set of *connector elements* that allow structural DSMLs to integrate with (or contain) CBML models
- Input ports directly connect to *Input Action* elements
- Output actions have the same name as their output port to reduce model clutter
  - i.e., prevent many to one connections



Code Generation for Emulation

Context

- The generation technique should not be dependent on the underlying technology
- Although we are targeting CUTS, we should be able to generate emulation code for any benchmarking framework



Generation Enablers

- *Emulation layer* – represents the application layer's "business logic" where elements in WML used to parameterize CBML behavior are mapped to this layer
- *Template layer* – acts as a bridge between the upper emulation layer & lower benchmarking layer to allows each to evolve independently of each other
- *Benchmark layer* – the actual benchmarking framework (e.g., CUTS)

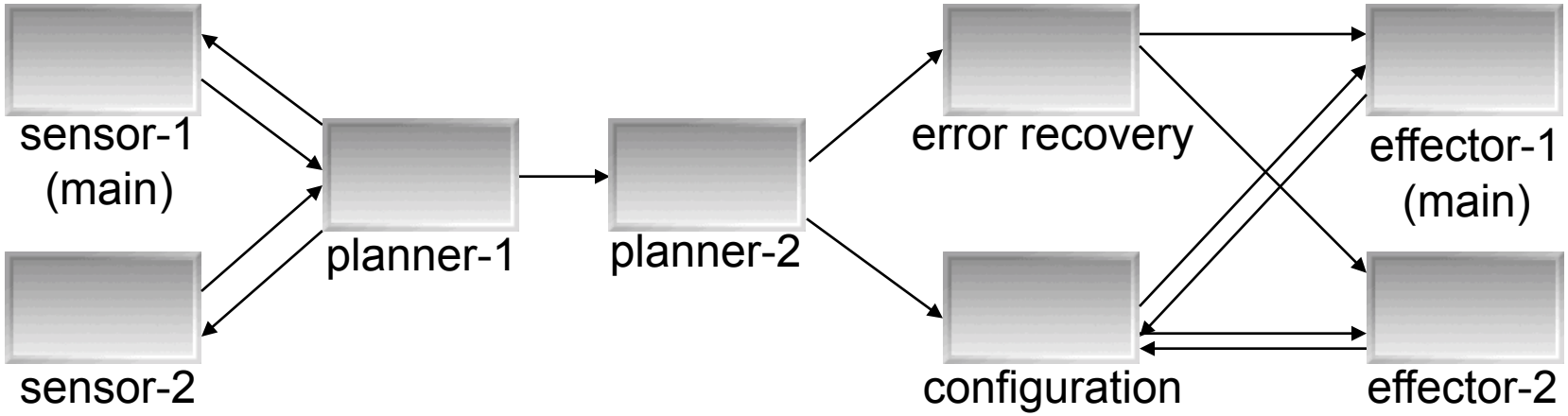
component method

template / benchmarking

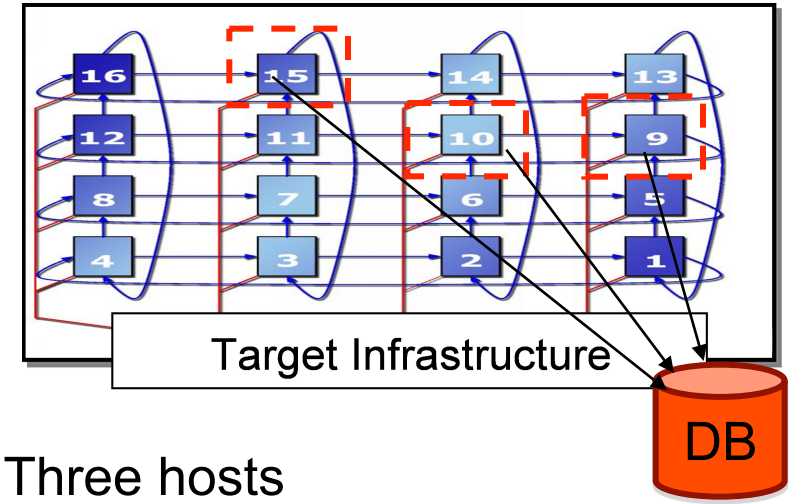
emulation

```
void DatabaseComponent::push_BasicType_Inpu
  Quer                                    ITH_DE
  ACE_                                    eptio
{
  // q                                    hread
  CUTS_                                   CUTS_THR    IVATION_RECORD ();
  this->basic_count_ ++

  record->perform_action_no_logging (
    CUTS_Database_Worker::query_stock_info (this->db_handle__));

  CUTS_CCM_Event_T <OBV_QueryResponse> __event_100000028__;
  this->context_->push_basic_response (__event_100000028__.in ());

  CUTS_CCM_Event_T <OBV_LogStatus> __event_100000029__;
  this->context_->push_log_status (__event_100000029__.in ());
}
```

Experiment: SEM Tools & SLICE Scenario (1/4)



Component Interaction for SLICE Scenario



- Three hosts
- One database is shared between all hosts

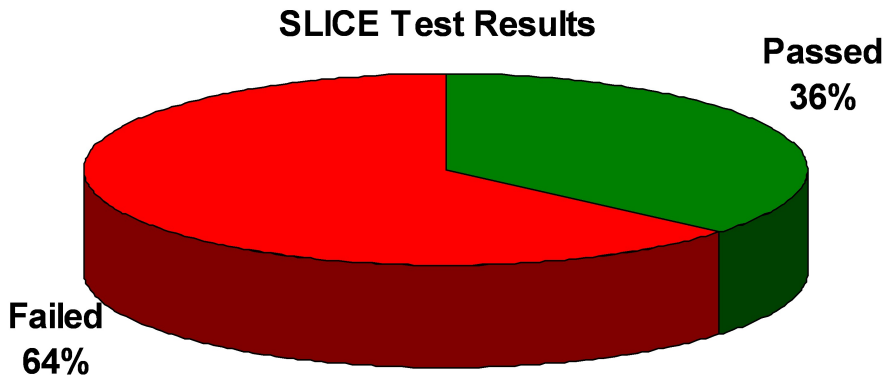
D&C & Performance Requirements

- *Critical path* deadline is 350 ms
  - main sensor to main effector through configuration
- Components in the critical paths must be deployed across all 3 hosts
- Main sensor & effector must be deployed on separate hosts

Experiment: SEM Tools & SLICE Scenario (2/4)

Evaluation Criteria	Description
(H1) Meeting Performance Requirements	We hypothesize that given the performance requirements of the SLICE scenario, we can locate D&Cs that meet the 350 ms critical path deadline
(H2) Early Understanding of System Performance	We hypothesize that the specified workload of the SLICE scenario is too much for a single host to handle, & meet the 350 ms critical path deadline

Experiment: SEM Tools & SLICE Scenario (3/4)

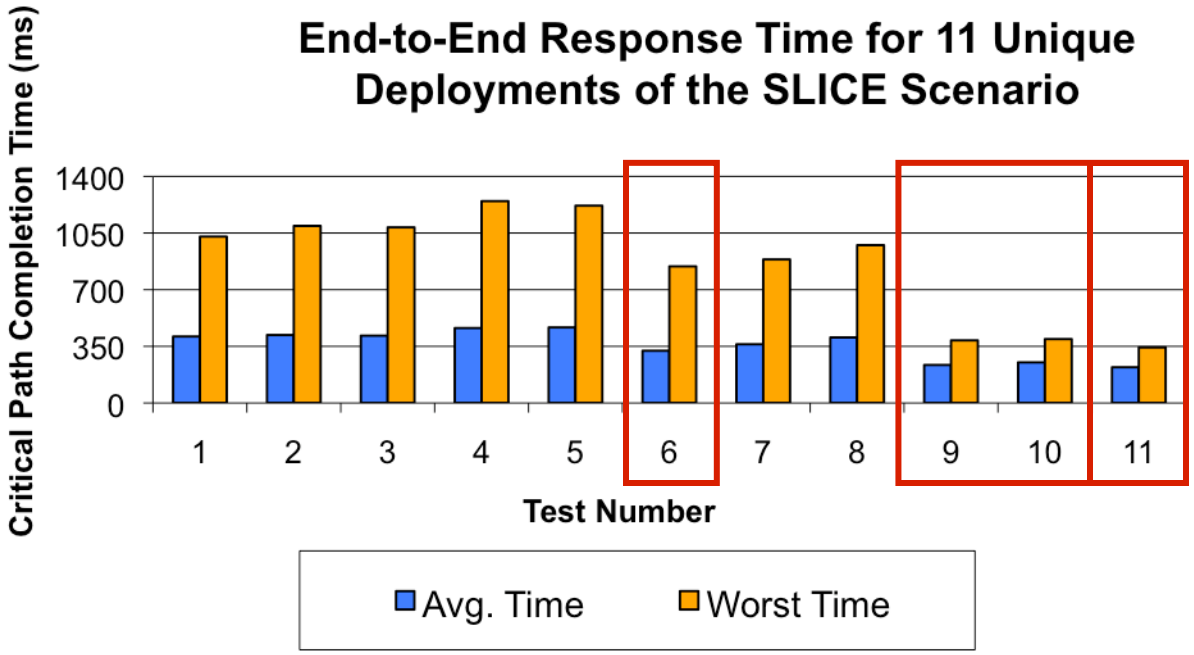


We are able to validate hypothesis 1 (H1) during the early stages of development on the target architecture

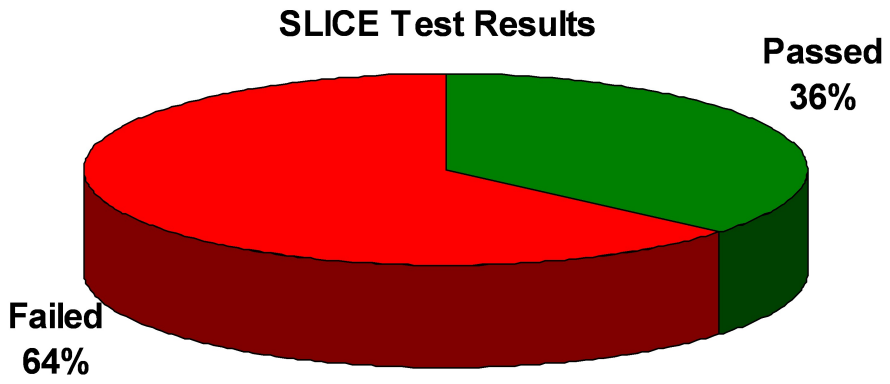
Population size of 11 tests

Hypothesis 1

- Only 4 of 11 deployments met the 350 ms critical path deadline for average time
- Test 11 the only test to meet critical path deadline for worst time
  - *e.g.*, least amount of timing contention



Experiment: SEM Tools & SLICE Scenario (3/4)

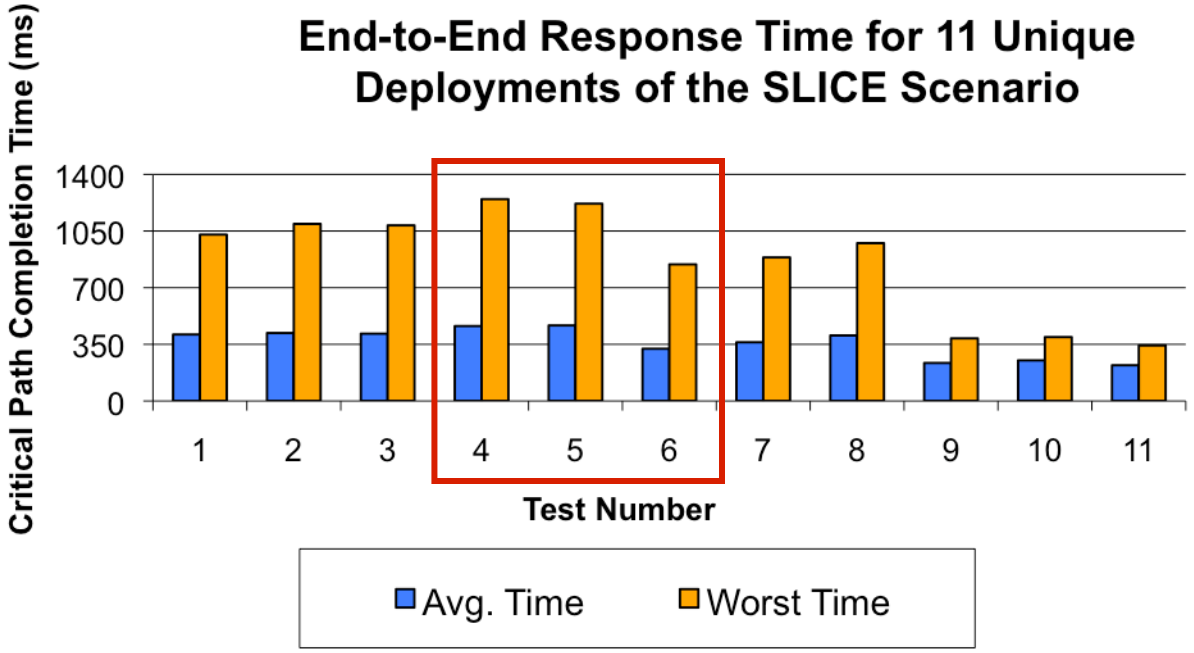


We are able to validate hypothesis 2 (H2) during the early stages of development on the target architecture

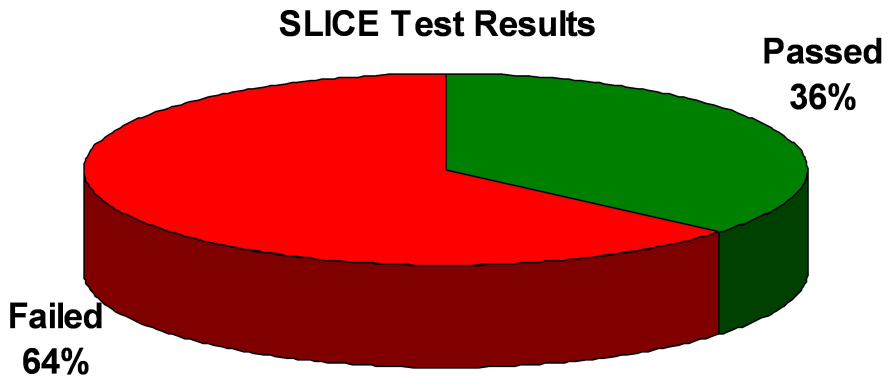
Population size of 11 tests

Hypothesis 2

- Test 4 deployed 6 of 7 components on the same host
- Test 5 deployed all the components on the same host
- Test 6 deployed critical path components on the same host, but violates deployment constraint

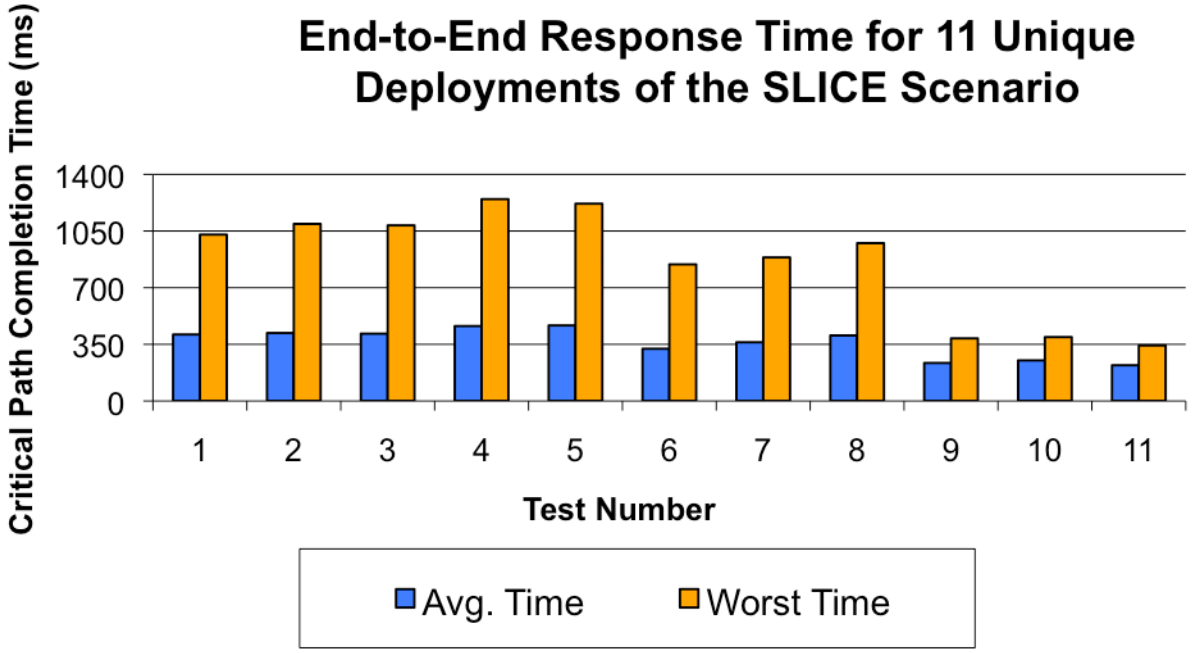


Experiment: SEM Tools & SLICE Scenario (4/4)



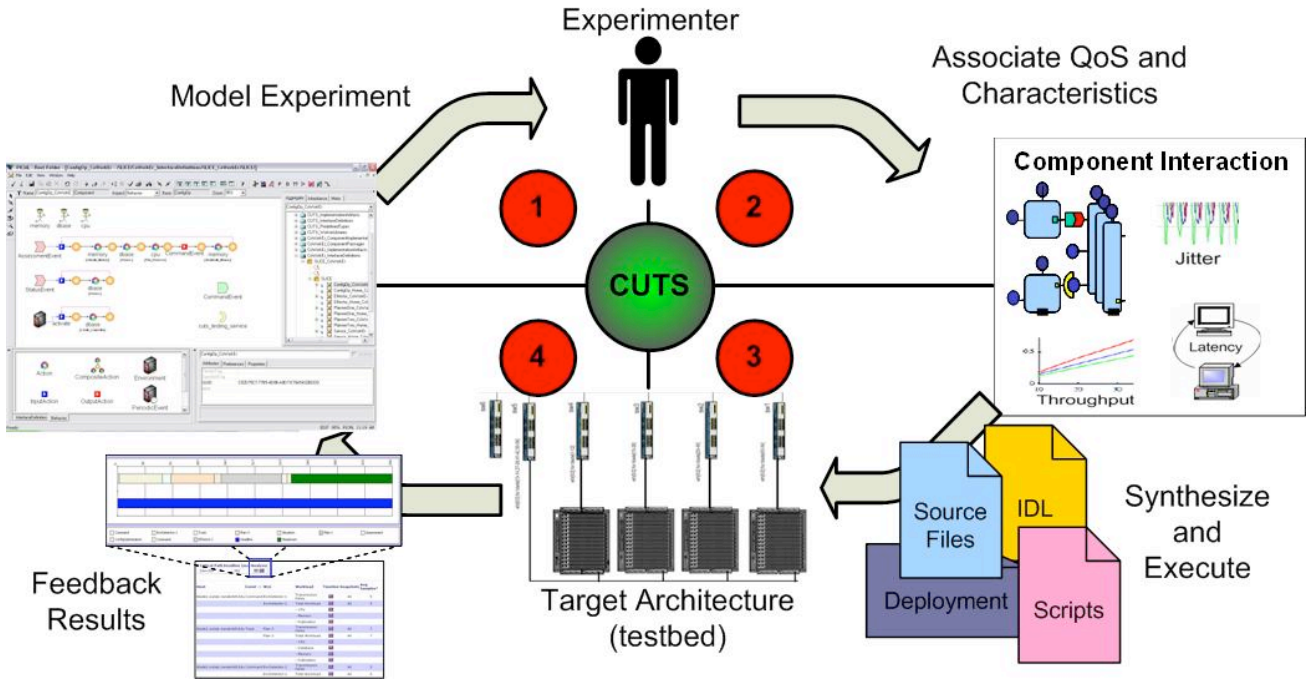
Population size of 11 tests

Conclusion: CUTS adapts BDD principles to evaluate non-functional concerns (*e.g.,* deployment) of DRE component-based systems during the early stages of development

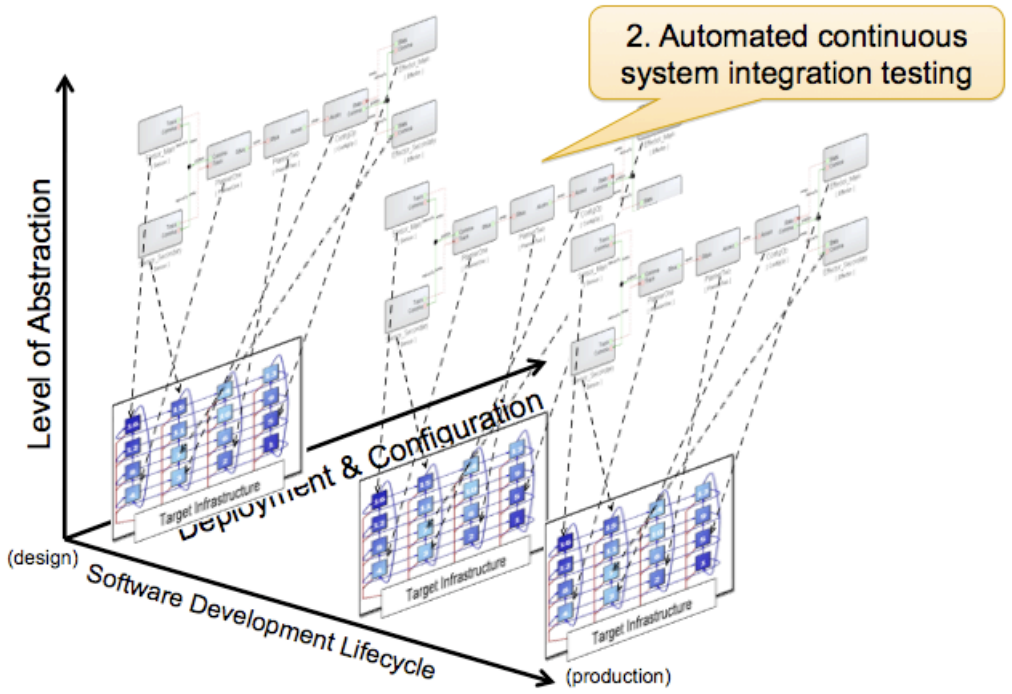


Coffee Break

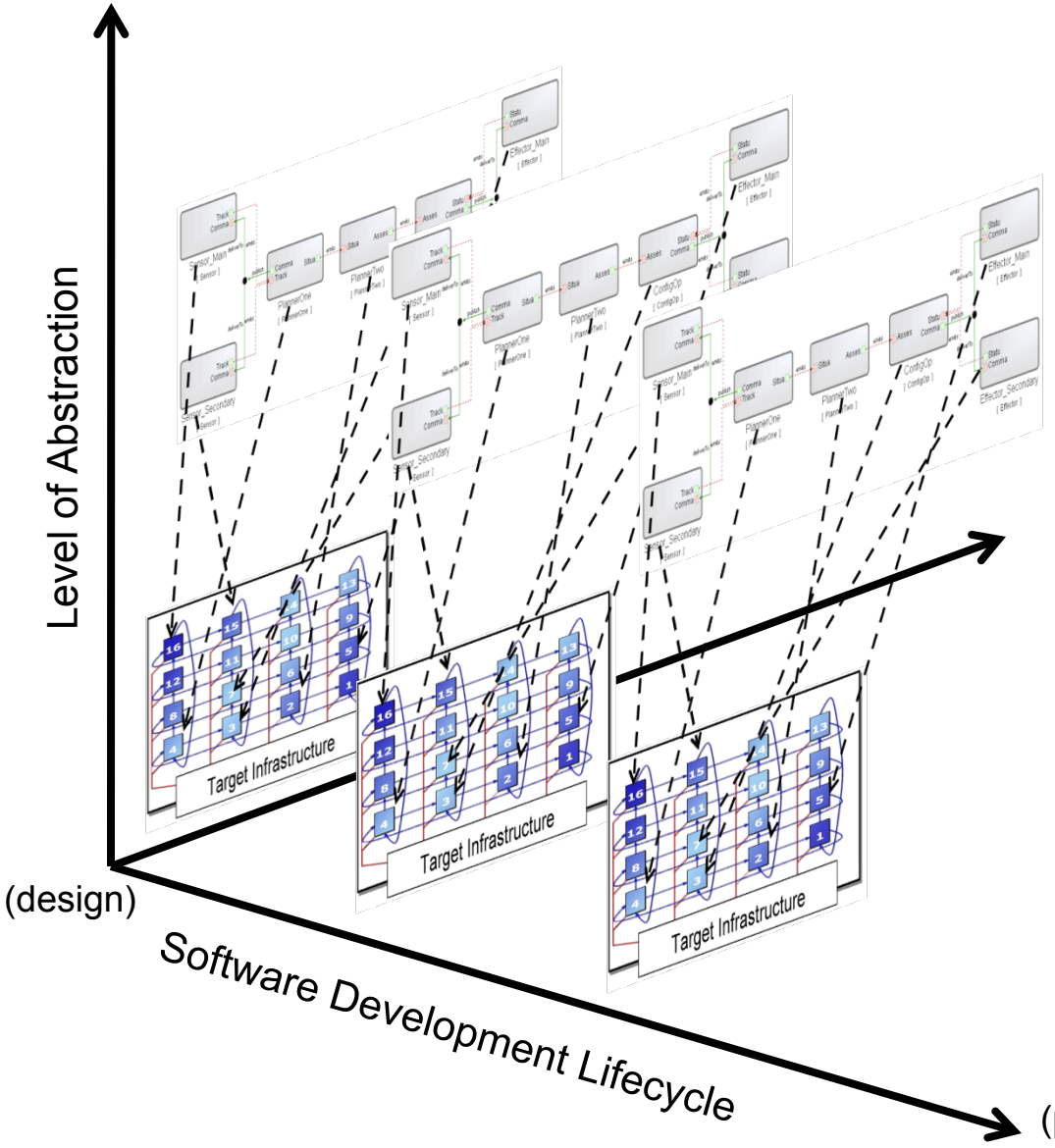
PART 4:
Hands-on Demo with CUTS



PART 5:
Continuous System Integration – Addressing Complexity #2



Continuous System Integration



As a system is developed, it should be tested continuously to ensure high-confidence in properties such as:

- The system is meeting its functional requirements
- The system's performance remains within its required specifications
  - *e.g.*, non-functional requirements
- The system can respond properly & in a timely manner to abnormal behavior

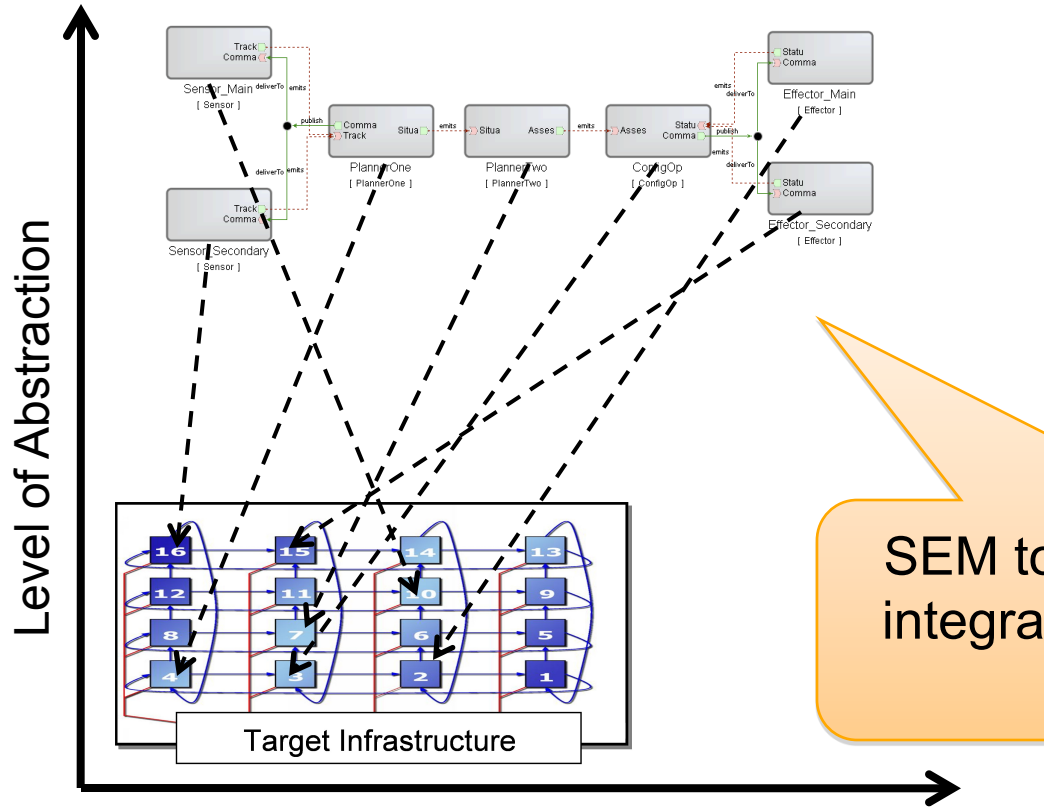
Continuous System Integration Properties

How do we realize continuous system integration of distributed component-based systems?

Property	Description
High-level Performance Goals	Technique provides means for capturing & analyzing high-level performance goals
Distributed Testing	Technique allows testing of distributed systems on their target architecture
Automated Testing	Testing technique supports automation
Continuous Testing	Technique continuously executes integration test throughout the development lifecycle, *e.g.*, after a commit to a source code repository

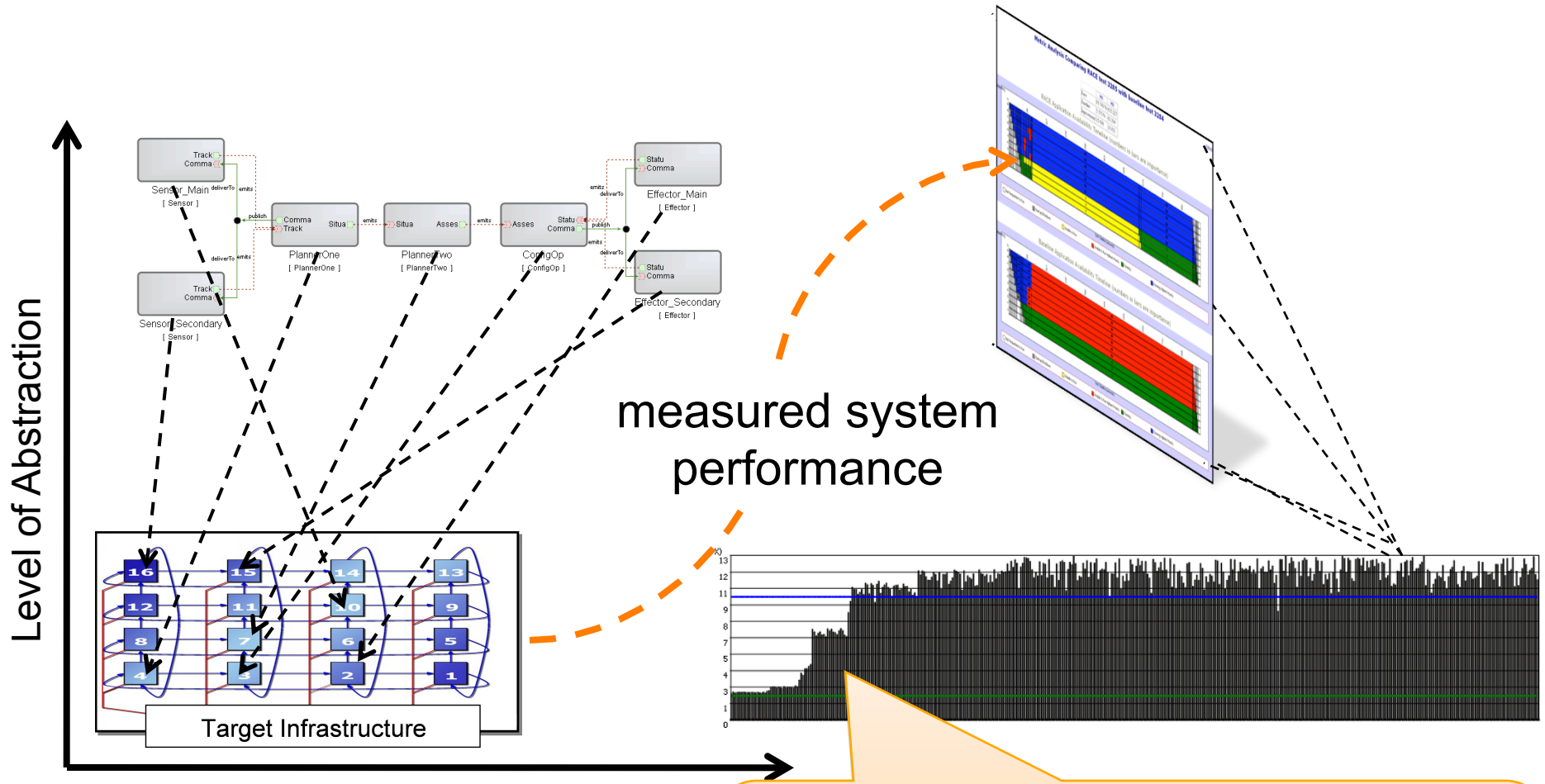
Note: Properties are based on characteristics of enterprise component-based systems & enterprises trying to apply agile development techniques on such systems

Challenge: Continuous Testing & Analysis



SEM tools enable early system integration testing on the target architecture

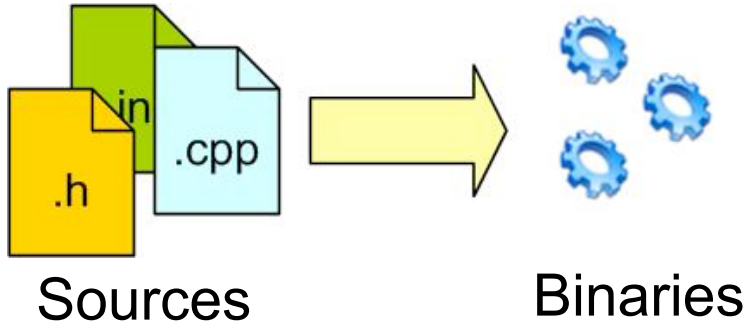
Challenge: Continuous Testing & Analysis



SEM tools focus on test generation, but do not provide efficient & effective capabilities for testing throughout the entire development lifecycle

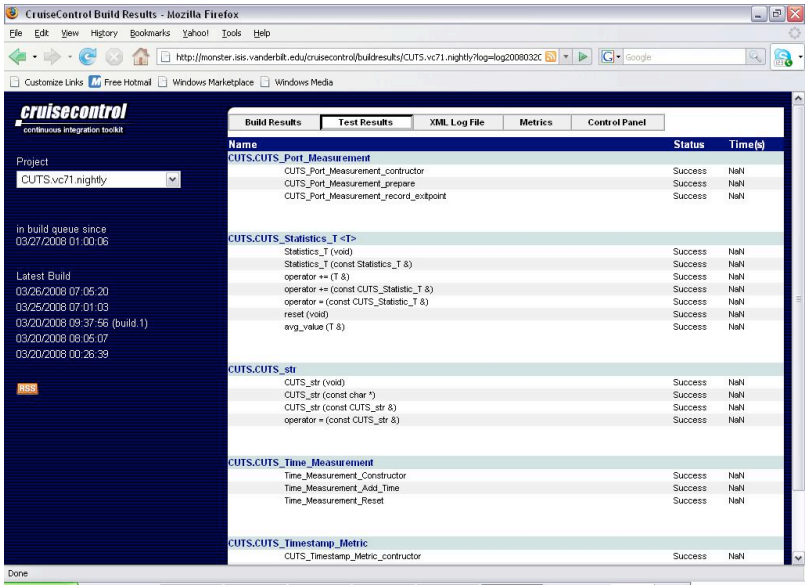
Integrate SEM Tools with Continuous Integration Environments

- Continuous integration environments continuously validate software quality by:
  1. performing automated builds commit or successful execution & evaluation of prior events,



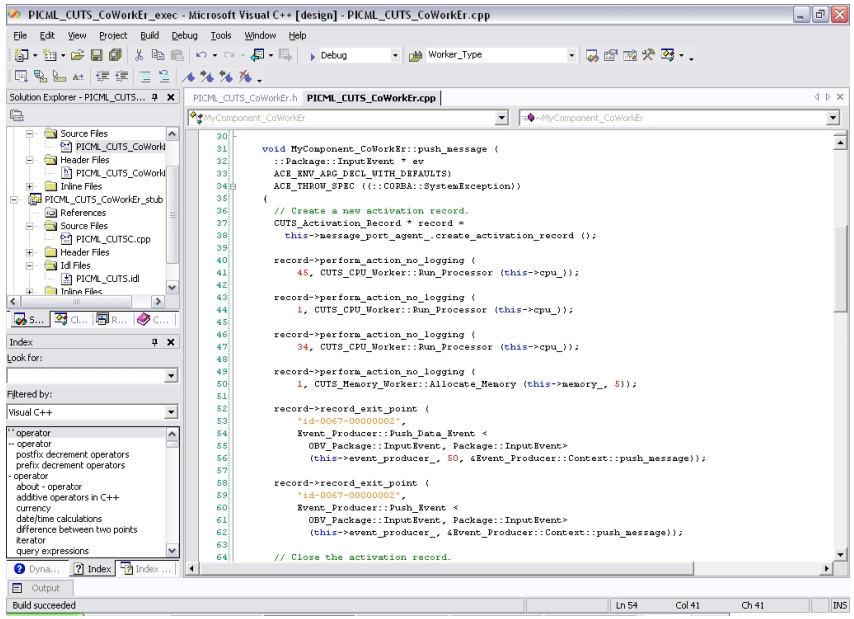
Integrate SEM Tools with Continuous Integration Environments

- Continuous integration environments continuously validate software quality by:
  1. performing automated builds commit or successful execution & evaluation of prior events,
  2. executing unit tests to validate basic system functionality,



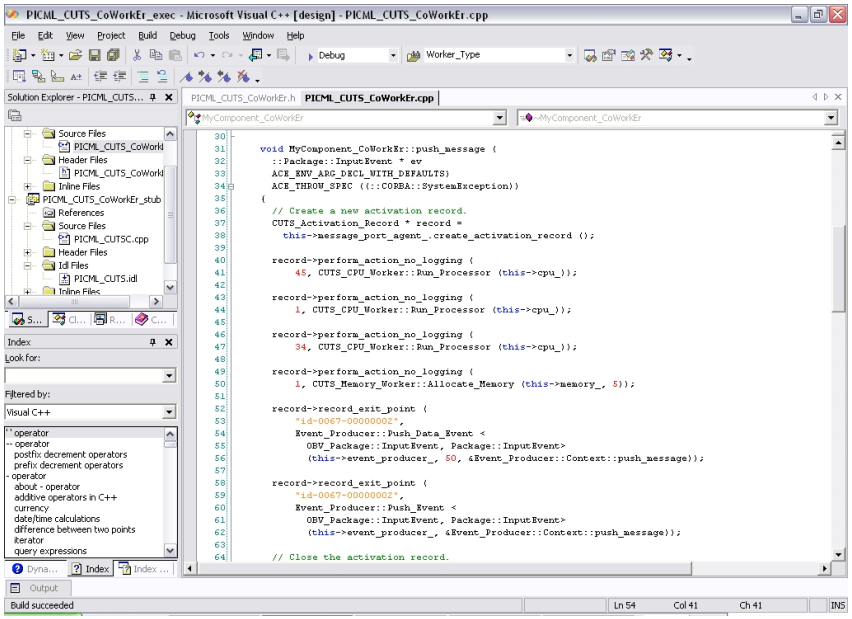
Integrate SEM Tools with Continuous Integration Environments

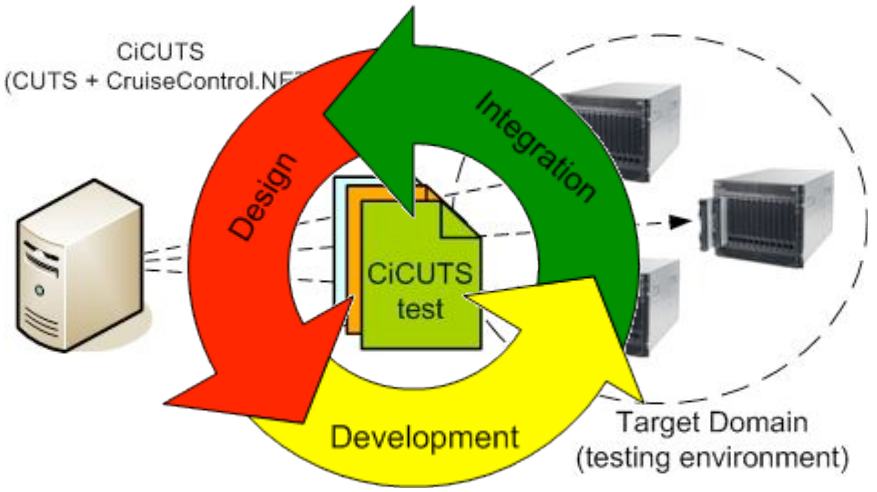
- Continuous integration environments continuously validate software quality by:
  1. performing automated builds commit or successful execution & evaluation of prior events,
  2. executing unit tests to validate basic system functionality,
  3. evaluating source code to ensure it meets coding standards, &
  4. executing code coverage analysis



Integrate SEM Tools with Continuous Integration Environments

- Continuous integration environments continuously validate software quality by:
  1. performing automated builds commit or successful execution & evaluation of prior events,
  2. executing unit tests to validate basic system functionality,
  3. evaluating source code to ensure it meets coding standards, &
  4. executing code coverage analysis
- CiCUTS (*i.e.*, combination of continuous integration environments with CUTS) uses integration tests managed by continuous integration environments that dictate the behavior of CUTS

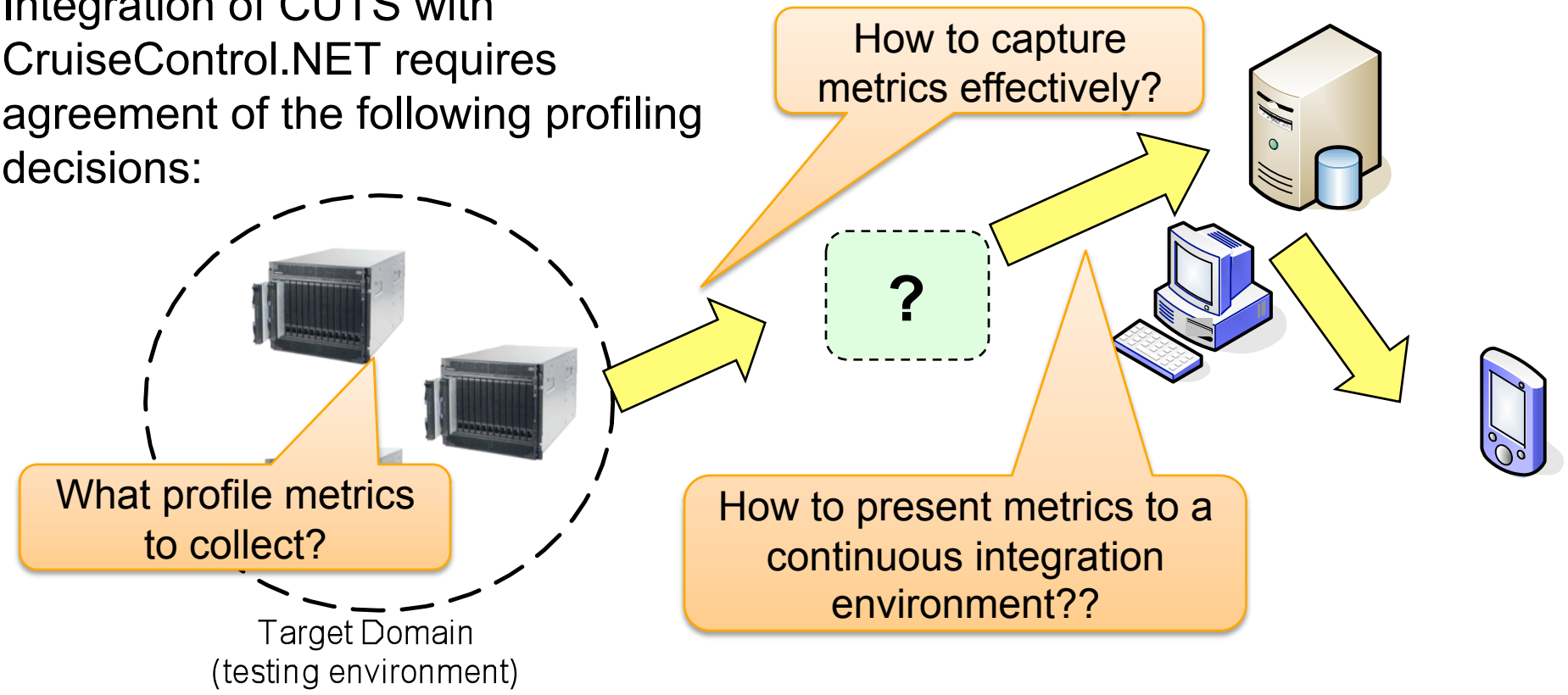




CiCUTS helps developers & testers ensure system QoS meets—or is close to meeting—its specification throughout the development lifecycle.

CiCUTS Integration Challenges

Integration of CUTS with CruiseControl.NET requires agreement of the following profiling decisions:



Integration Alternatives

- Extend profiling infrastructure of SEM tools to capture domain-specific metrics
- Capture domain-specific performance metrics in format understood by continuous integration environments
- Capture domain-specific performance metrics in an intermediate format

Alternative 1: Extending Profiling Infrastructure

Context

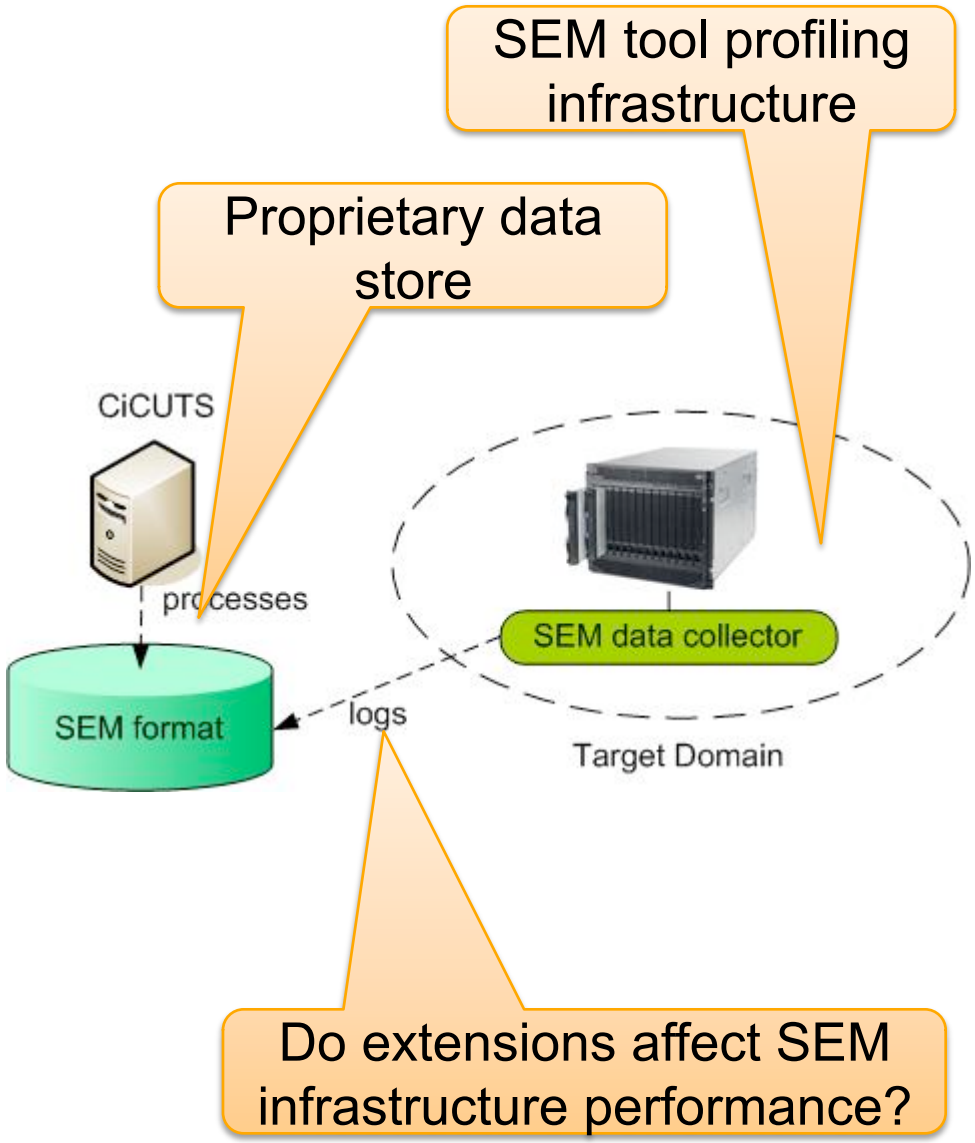
- SEM tools provide profiling infrastructures to collect predefined performance metrics
  - *e.g.*, execution times of events/ function calls or values of method arguments

Advantages

- Simplifies development of profiling framework
  - *e.g.*, can leverage existing data collection techniques

Disadvantages

- Must ensure domain-specific metrics do not effect existing SEM tool performance
- SEM tools may be proprietary & extension may be prohibited



Alternative 2: Capture Metrics In Format Understood By Continuous Integration Environment

Context

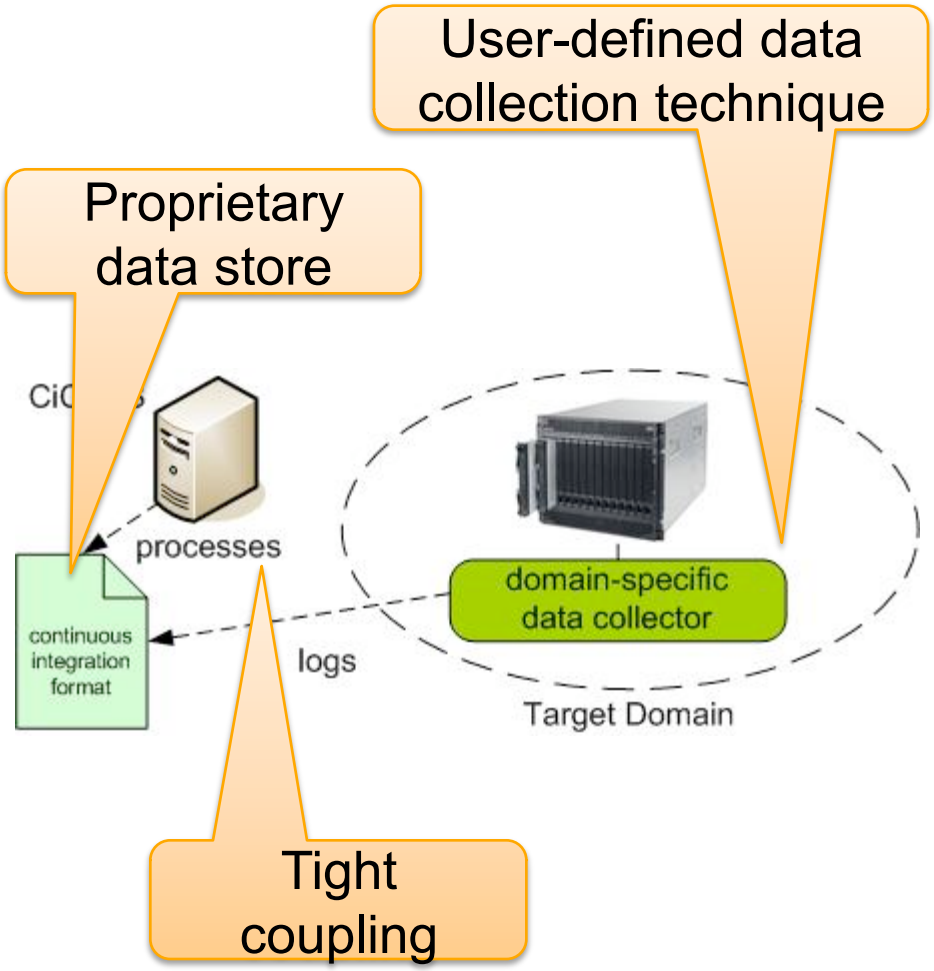
- Continuous integration environments use proprietary formats to store & analyze data
- May be feasible to collect & present metrics in format understood by continuous integration environments

Advantages

- Simplifies integration at the continuous integration side since format is known a priori

Disadvantages

- Requires a custom testing framework (adapter) to present data
- Tightly couples SEM tool with continuous integration environment



Alternative 3: Capture Metrics In Intermediate Format

Context

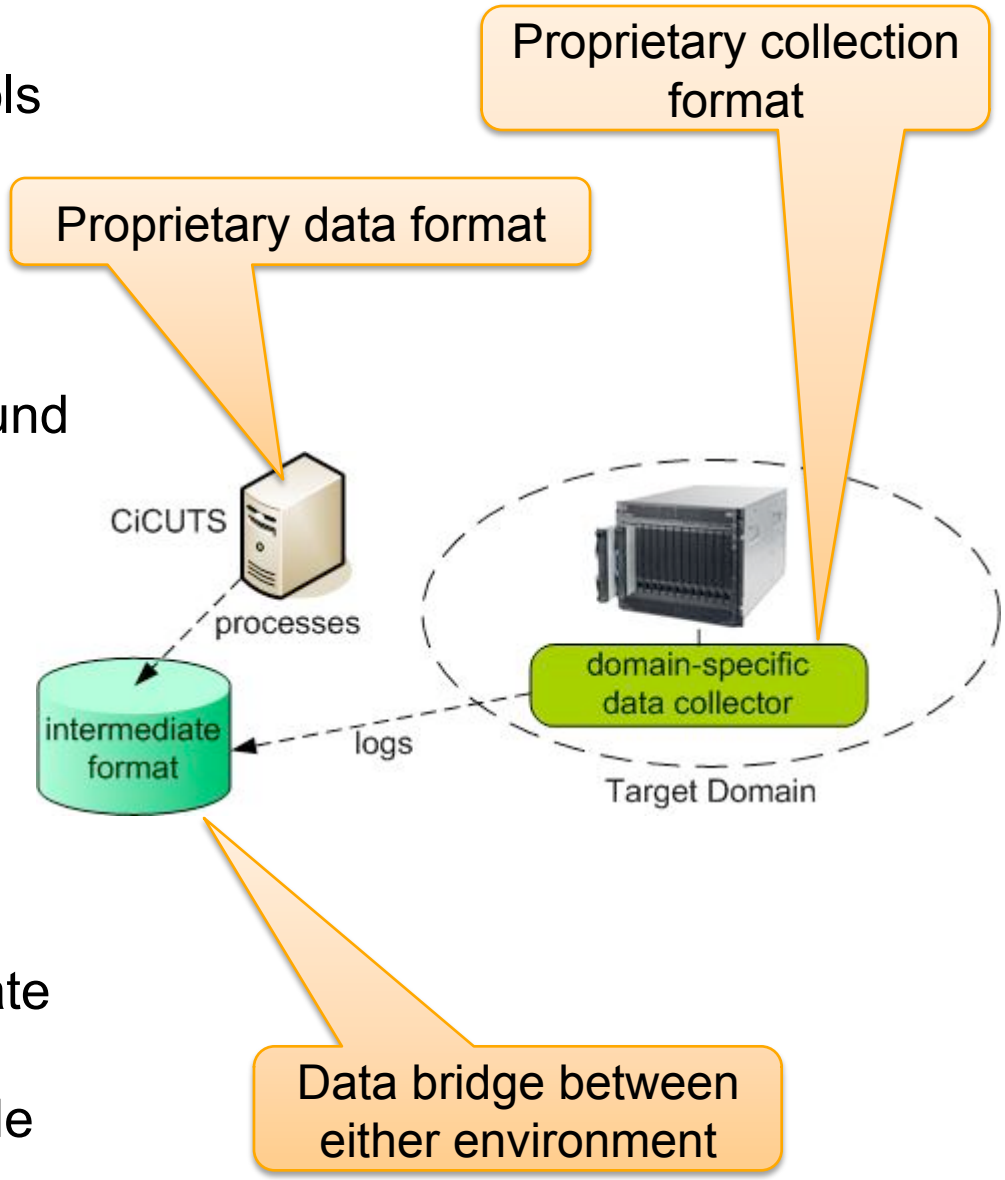
- Continuous integration & SEM tools each have proprietary methods
  - *e.g.*, data collection & representation
- May be feasible to store data in intermediate format that is not bound to a specific tool

Advantages

- Decouples continuous integration environment from the SEM tool
- Collection can be transparent to existing SEM tool infrastructure

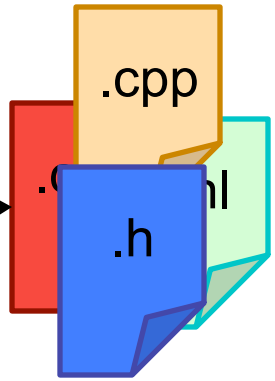
Disadvantages

- Requires agreement of intermediate format & implementation of data collectors & adapters on either side of the data store

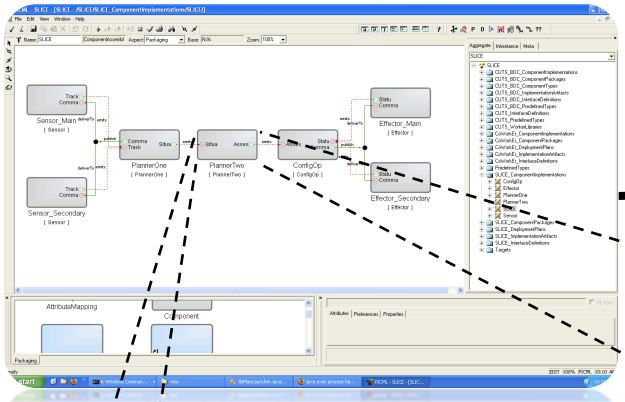


Solution: SEM Tools + Continuous Integration

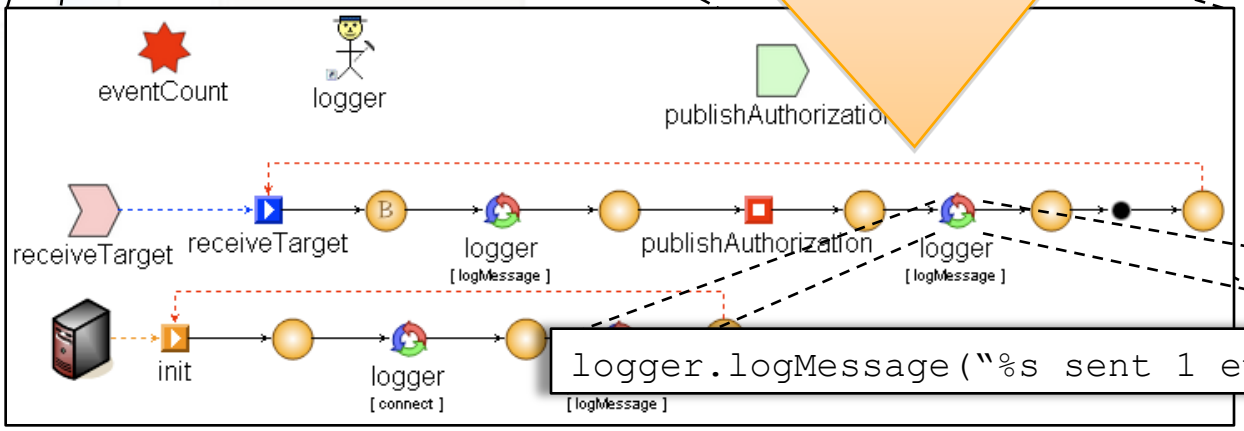
- CiCUTS uses integration alternative 3 because of its decoupled design feature
  - *e.g.*, developers can select different integration systems or SEM technologies, but leverage same technique



generates



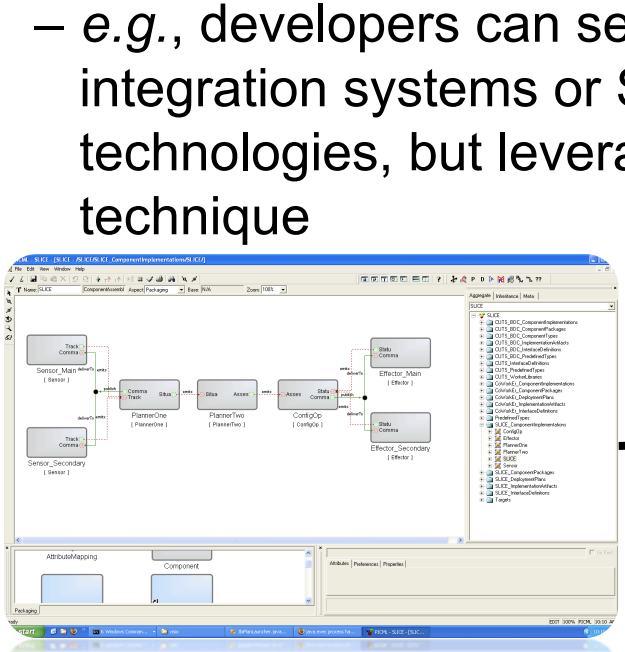
Simple log messages make it **easy to capture metrics** of interest & can be **updated to handle changing system requirements**



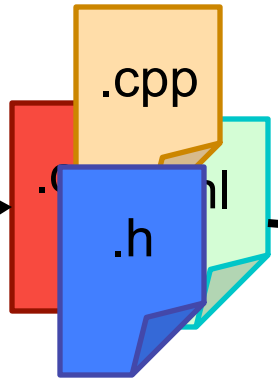

```
logger.logMessage("%s sent 1 event",instance_name);
```

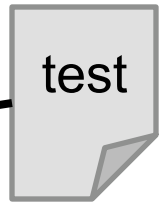
Solution: SEM Tools + Continuous Integration

- CiCUTS uses integration alternative 3 because of its decoupled design feature
  - *e.g.*, developers can select different integration systems or SEM technologies, but leverage same technique



generates

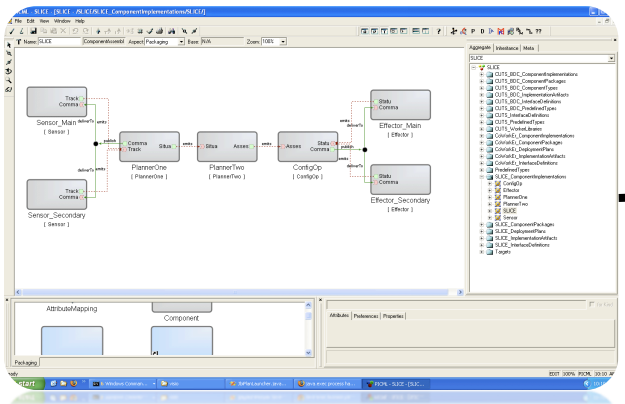




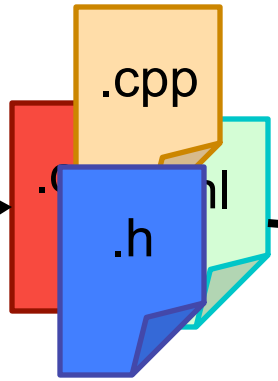
Without continuous integration service, developers must **manually manage** a distributed testing environment

Solution: SEM Tools + Continuous Integration

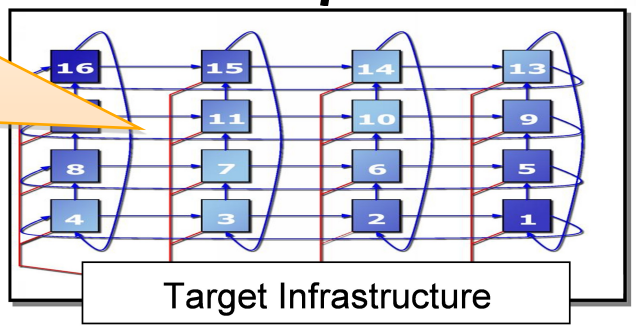
- CiCUTS uses integration alternative 3 because of its decoupled design feature
  - *e.g.*, developers can select different integration systems or SEM technologies, but leverage same technique



generates



test

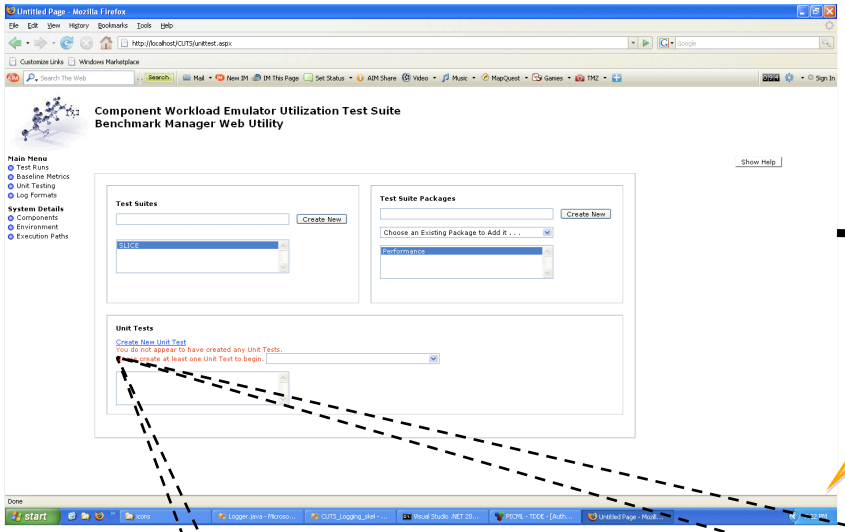


The distributed testing environment makes it **easy to automatically collect, store, & correlate metrics** from many tests

Solution: SEM Tools + Continuous Integration

- CiCUTS uses integration alternative 3 because of its decoupled design feature
  - *e.g.*, developers can select different integration systems or SEM technologies, but leverage same technique

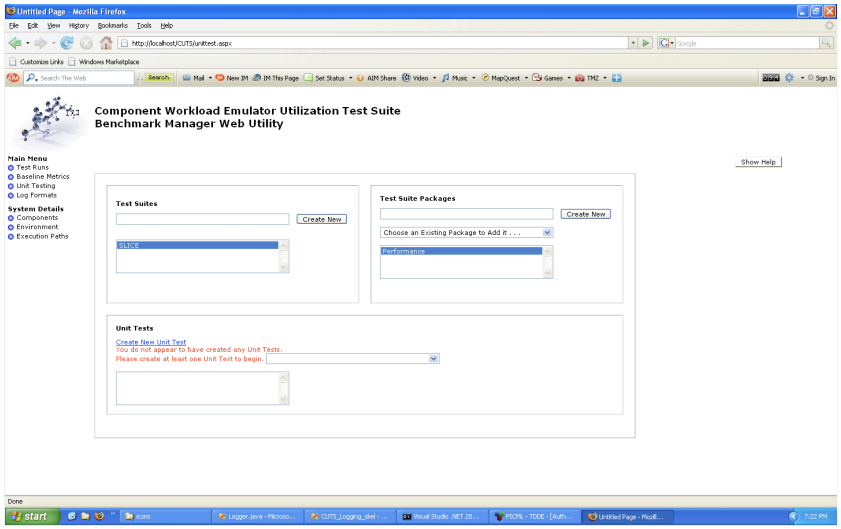
High-level constructs identify & formulate equations based on identified metrics &allow developers to analyze performance at **same level as system specification**



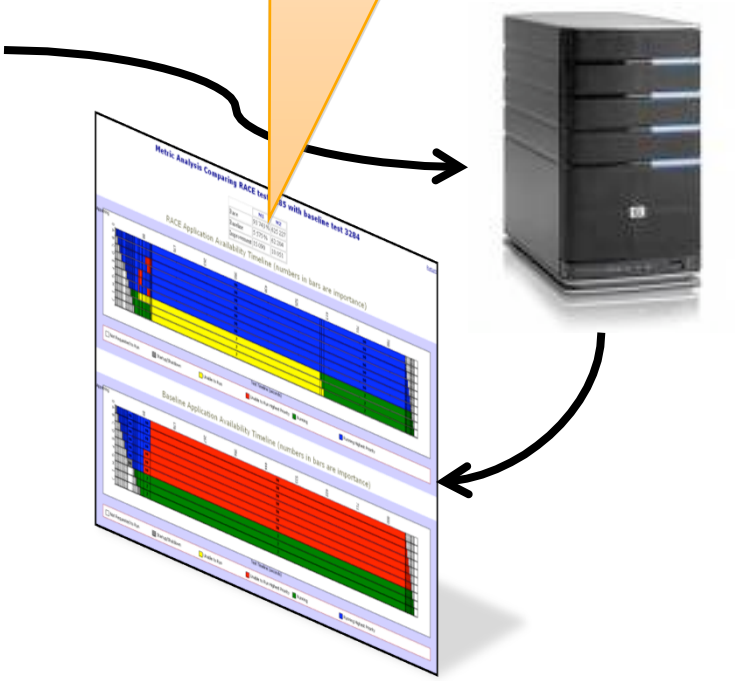
```
{STRING instance} sent {INT count} event
```

Solution: SEM Tools + Continuous Integration

- CiCUTS uses integration alternative 3 because of its decoupled design feature
  - *e.g.*, developers can select different integration systems or SEM technologies, but leverage same technique

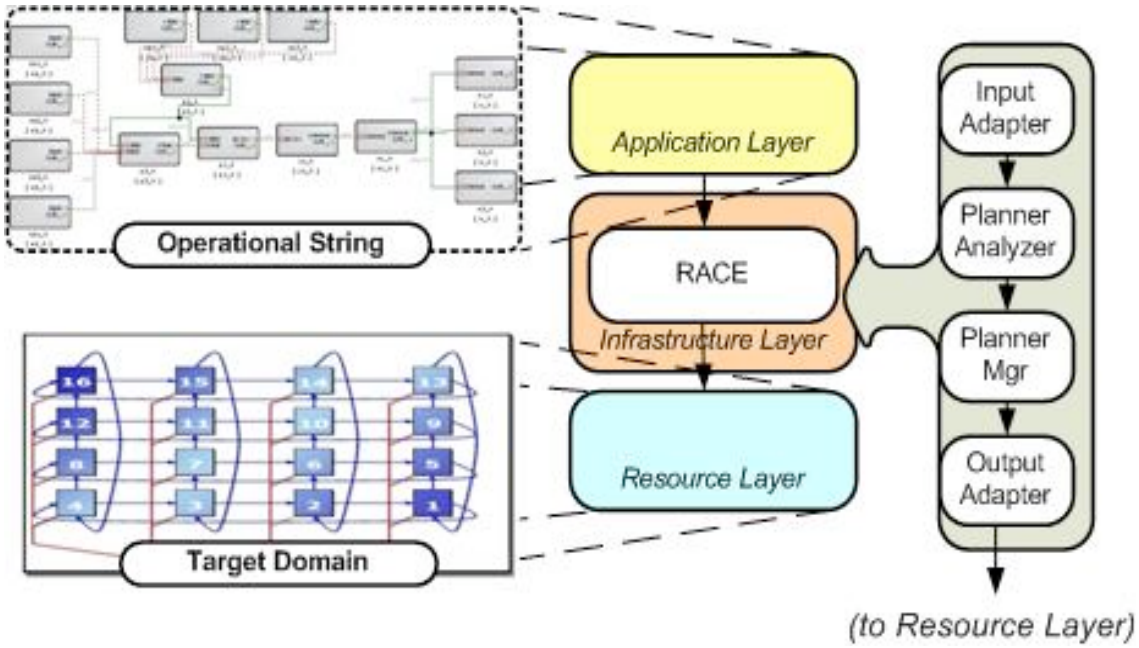


Without high-level constructs, developers would not be able to view **high-level performance graphs** of system



Experiment: SEM + CI & RACE Baseline Scenario

- RACE is a component-based system that manages workflows deployments
- RACE supports two types of workflow deployments
  - **Static** – deployments created offline where components are assigned to hosts
  - **Dynamic** – deployments created online, but component assignment to host is based on operating conditions

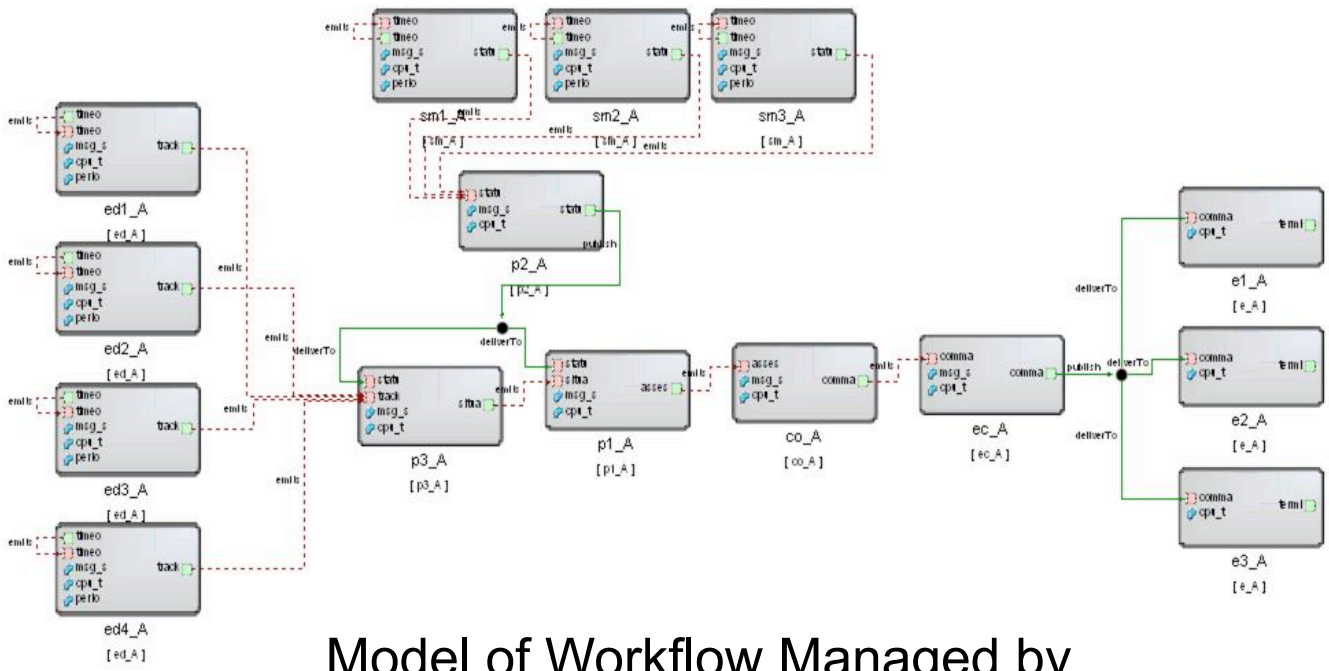


- *Baseline scenario* - higher priority workflows must have longer lifetime than lower priority workflows
  - e.g., under low resource availability

Experiment: SEM + CI & RACE Baseline Scenario

Experiment Design

- Constructed 10 identical workflows with different importance values & used CUTS to generate implementation
  - Workflow A – H: 90
  - Workflow I – J: 2
- Updated RACE source code with log messages that contained metrics of interest
- Constructed tests to be managed by the continuous integration server



Model of Workflow Managed by RACE

Experiment: SEM + CI & RACE Baseline Scenario

Evaluation Criteria	Description
(H1) Evaluating High-level QoS Requirements	We hypothesize that RACE can improve the lifetime of workflows deployed dynamically by at least 10% vs. workflows deployed statically

Results: Single Test Performance

Metric Analysis Comparing RACE test 3285 with baseline test 3284

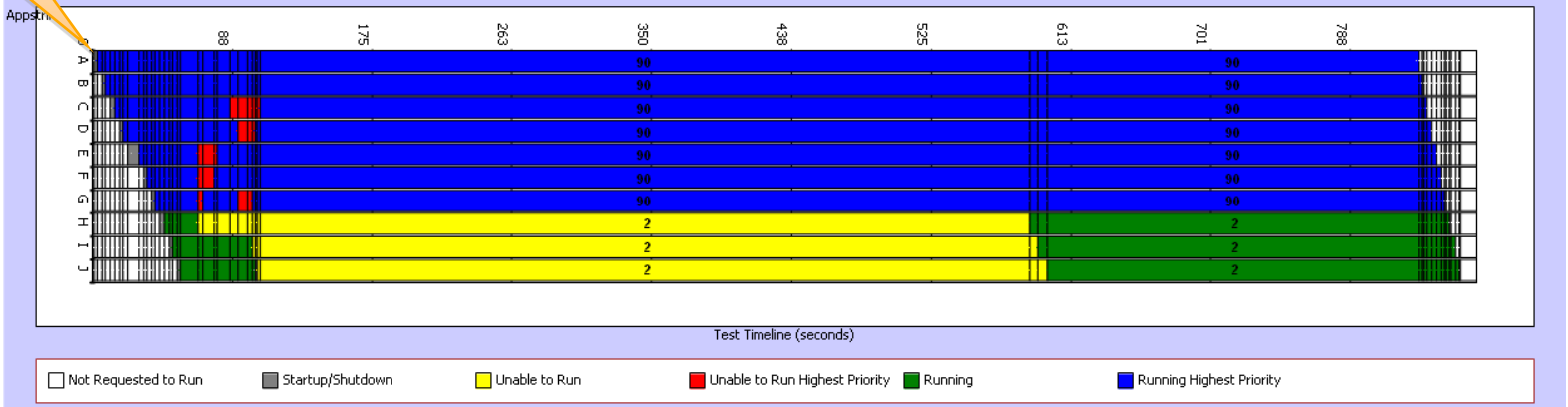
Return

	M1	M2
Race	93.743 %	625.227
Baseline	5.575 %	62.204
Improvement	15.090	10.051

Start of test

Dynamic Deployment Log Message Reconstruction

RACE Application Availability Timeline (numbers in bars are importance)



Start of test

Static Deployment Log Message Reconstruction

Baseline Application Availability Timeline (numbers in bars are importance)

Results: Single Test Performance

Kill node with higher importance operational string

Dynamic Deployment Log Message Reconstruction

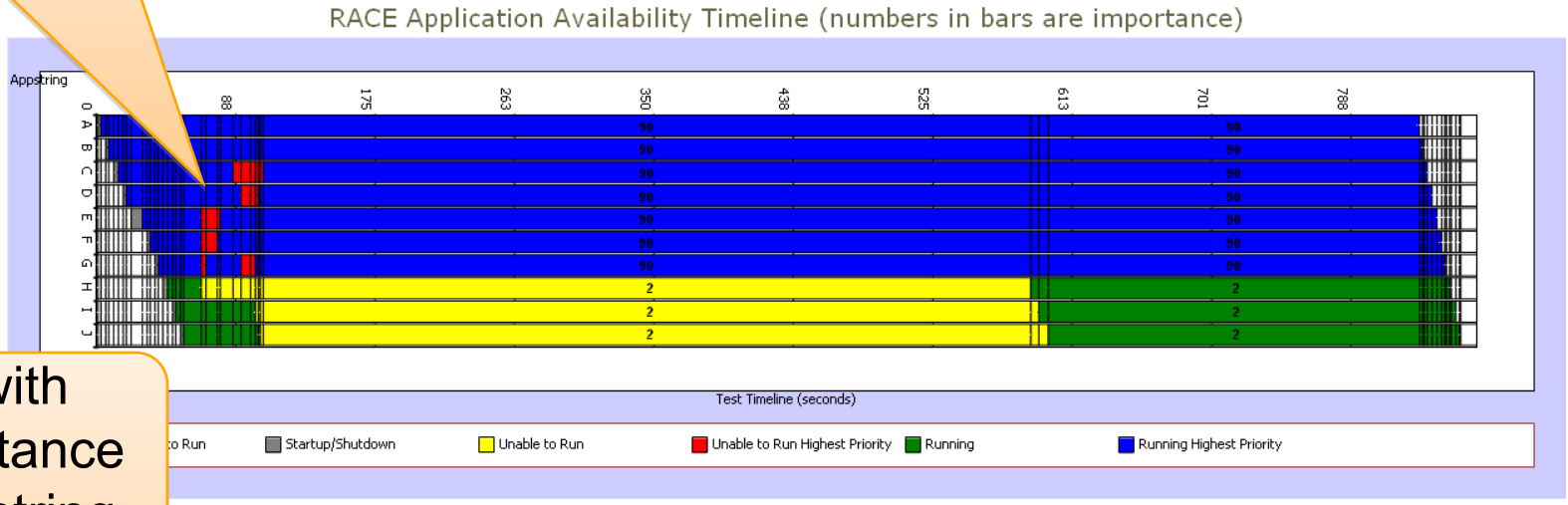
Kill node with higher importance operational string

Static Deployment Log Message Reconstruction

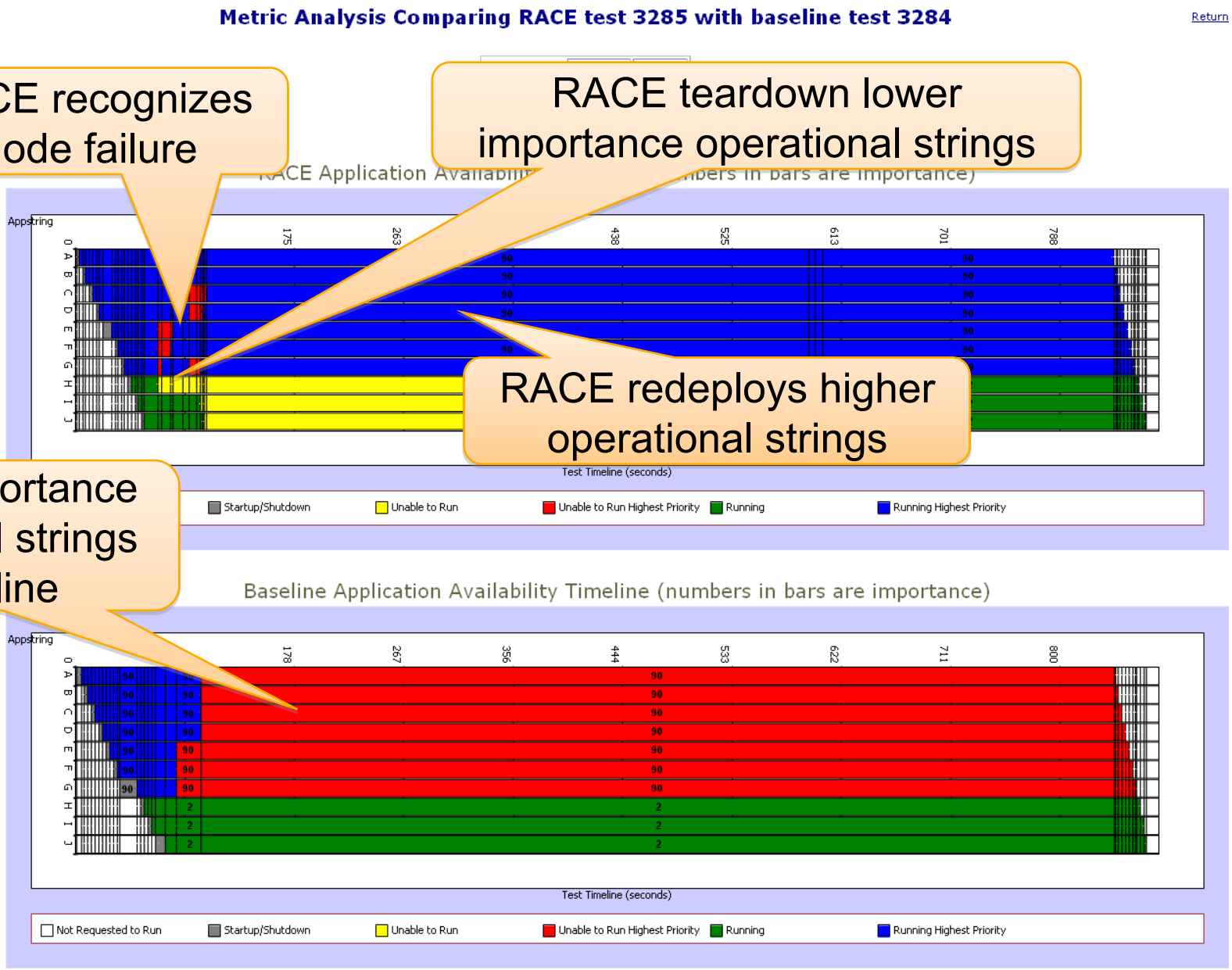
Metric Analysis Comparing RACE test 3285 with baseline test 3284

Return

	M1	M2
Race	93.743 %	625.227
Baseline	5.575 %	62.204
Improvement	15.090	10.051



Results: Single Test Performance



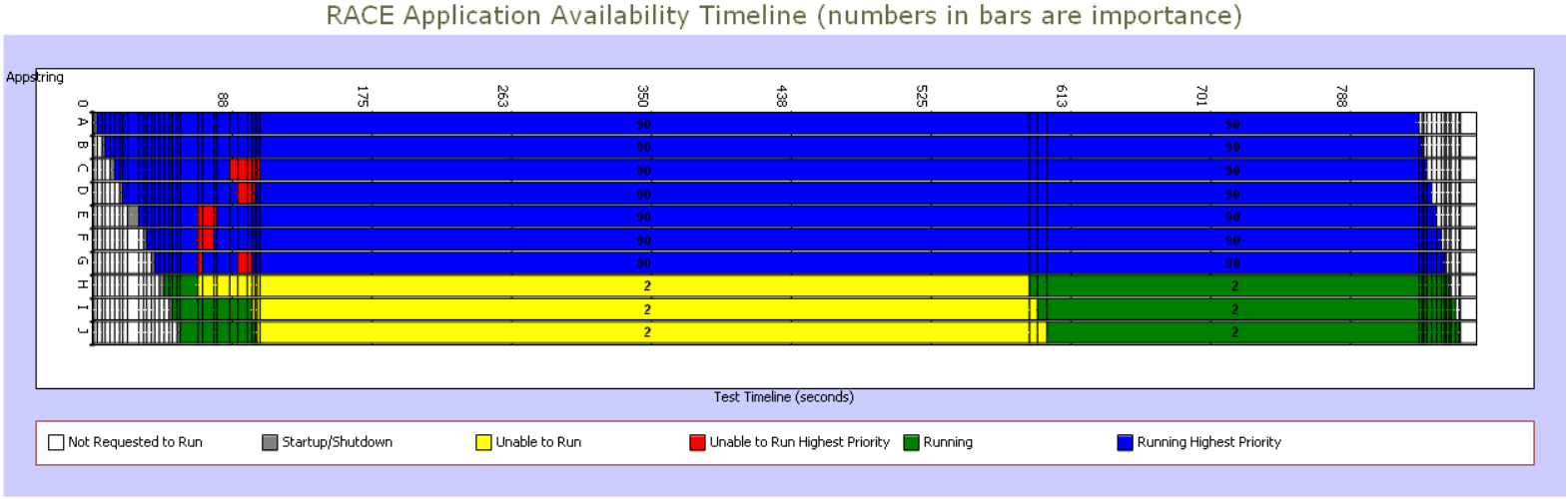
Results: Single Test Performance

Metric Analysis Comparing RACE test 3285 with baseli

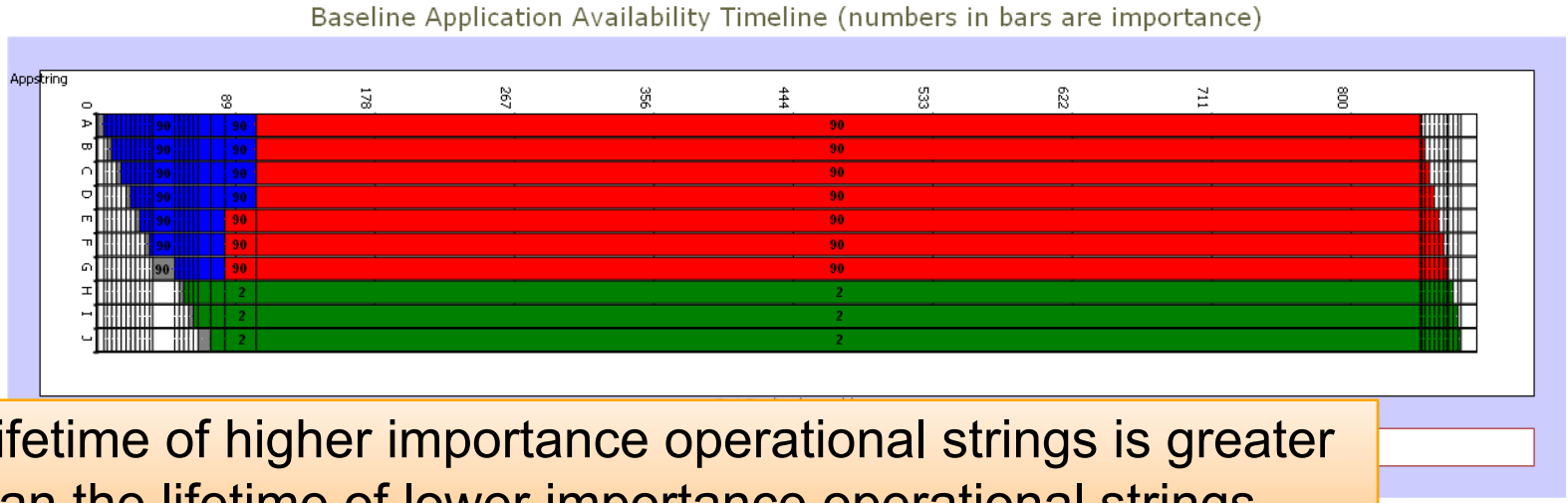
	M1	M2
Race	93.743 %	625.227
Baseline	5.575 %	62.204
Improvement	15.090	10.051

Improvement of dynamic vs. static

Dynamic Deployment Log Message Reconstruction

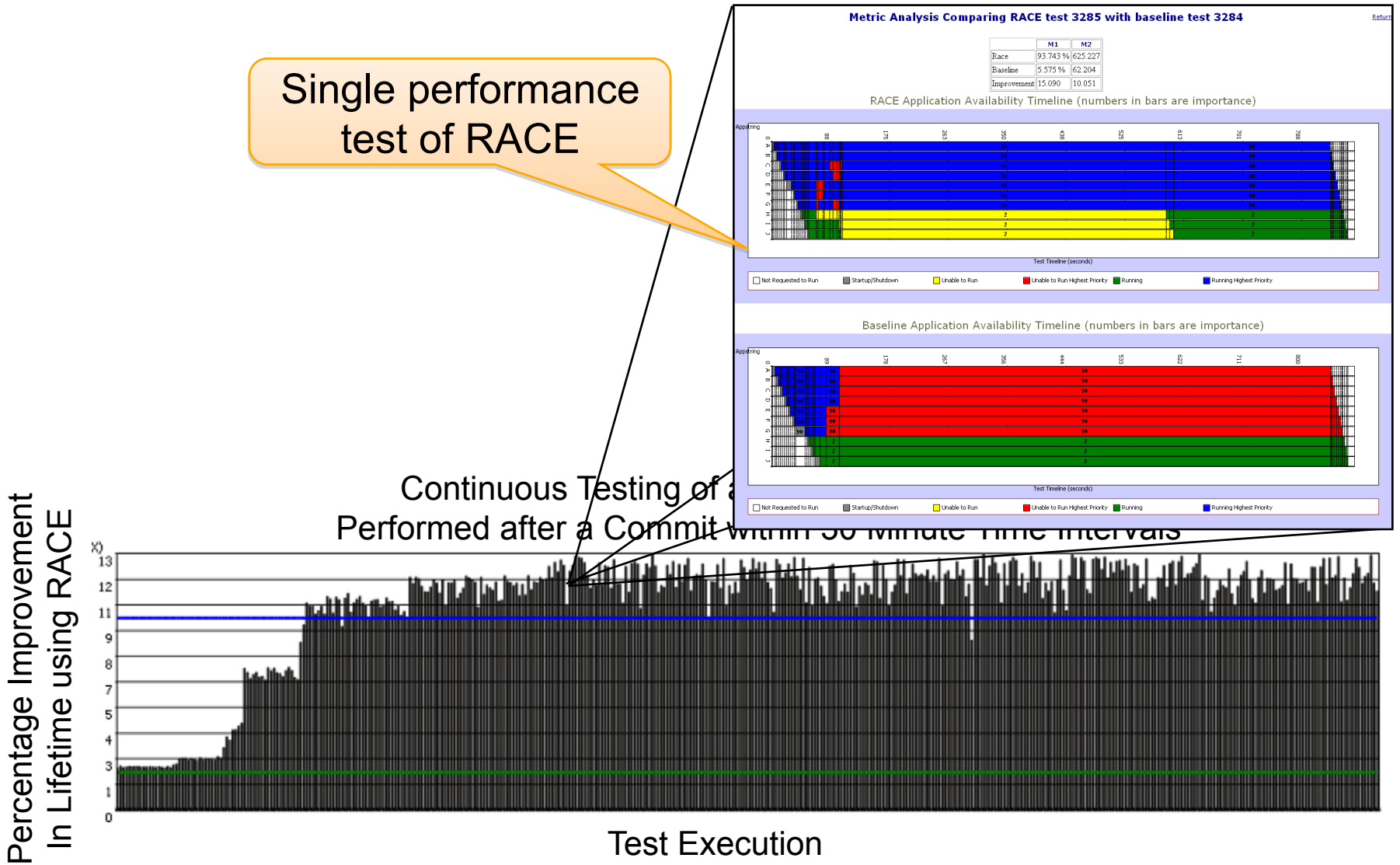


Static Deployment Log Message Reconstruction

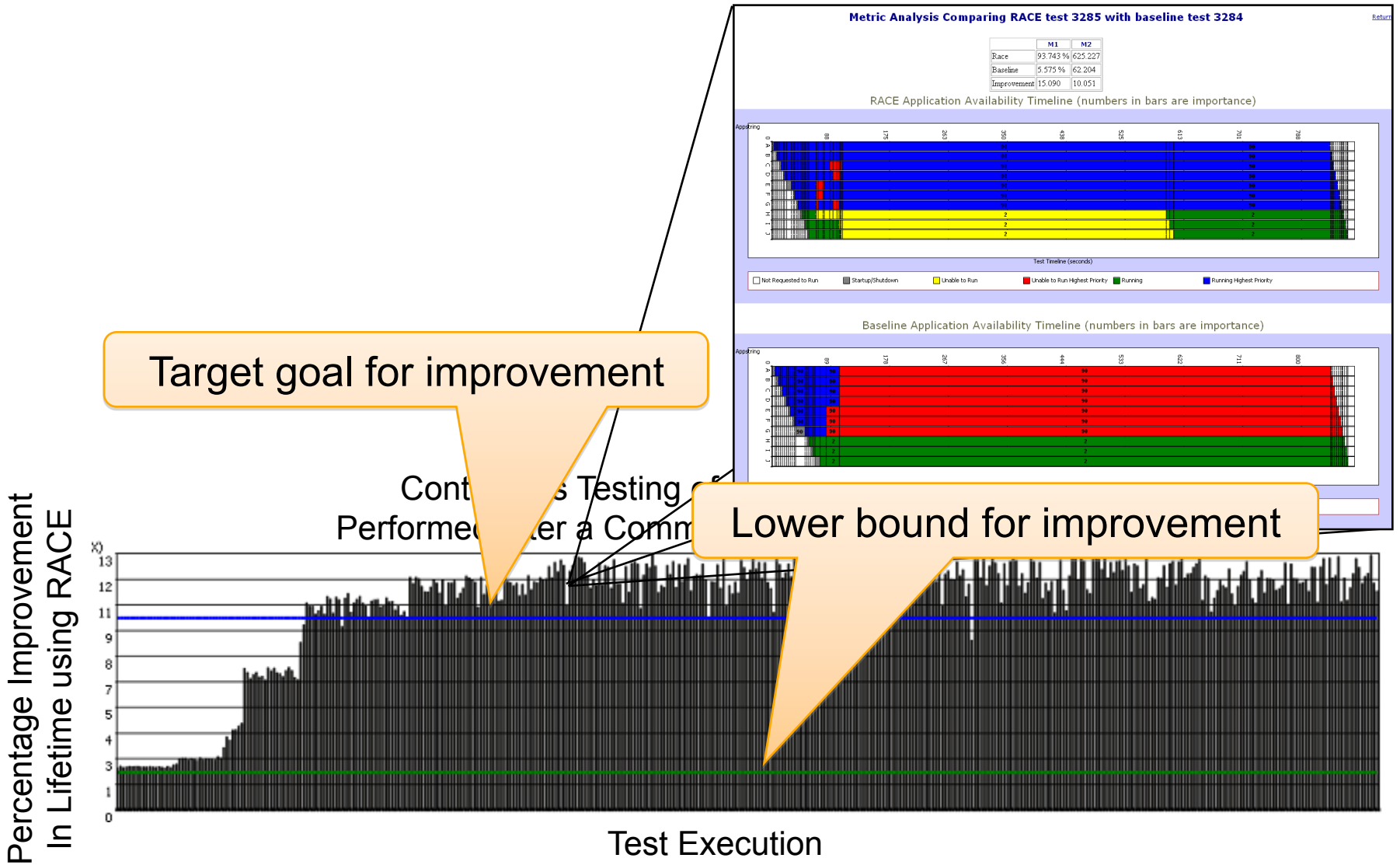


The lifetime of higher importance operational strings is greater than the lifetime of lower importance operational strings

Results: Multiple Tests Performance

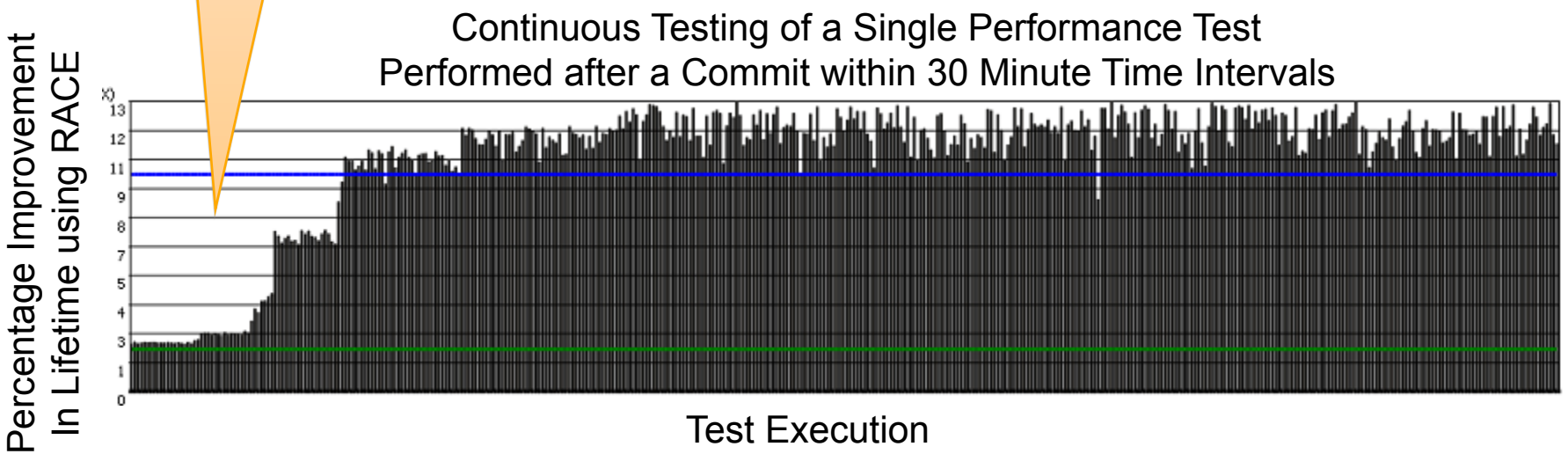


Results: Multiple Tests Performance

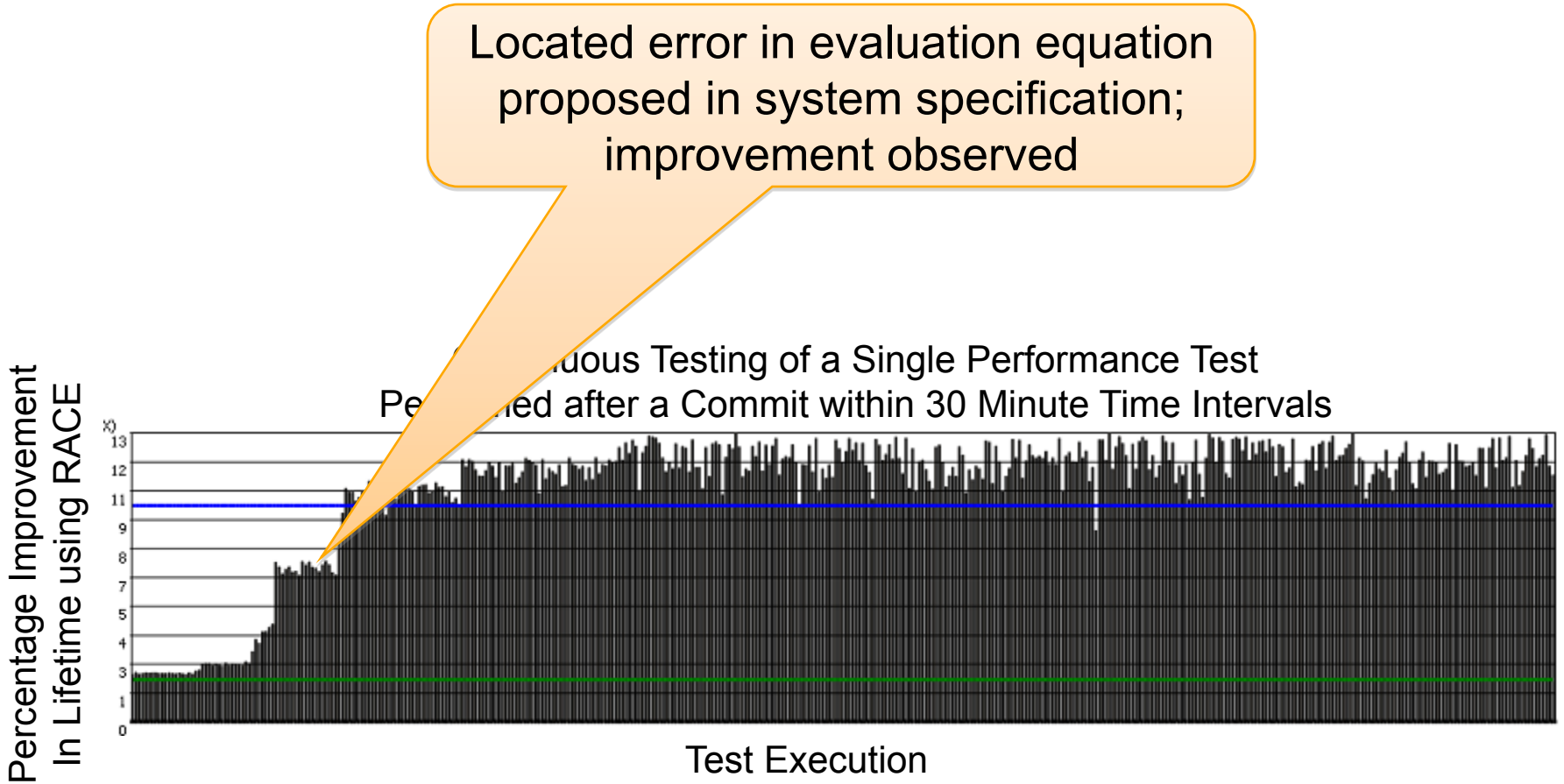


Results: Multiple Tests Performance

Initial development efforts are below hypothesized improvement

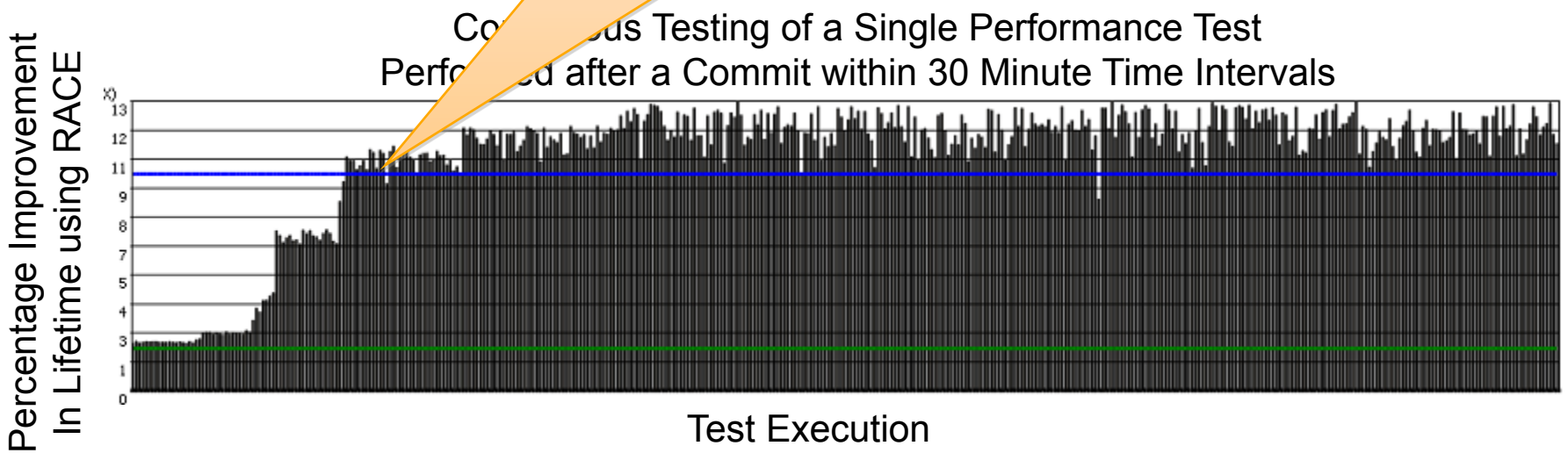


Results: Multiple Tests Performance



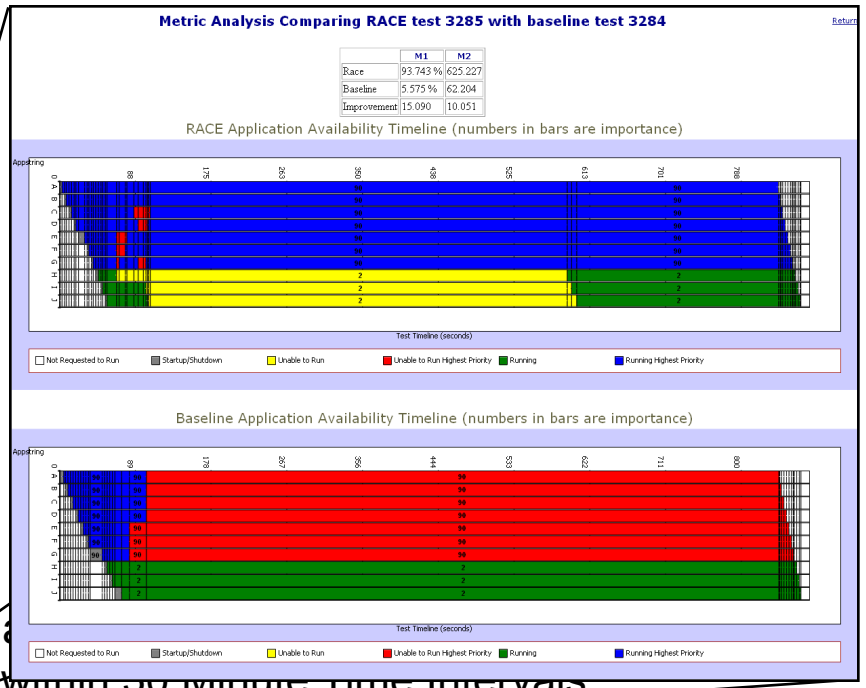
Results: Multiple Tests Performance

Improved RACE's implementation & observed another increase in performance; meeting system requirements

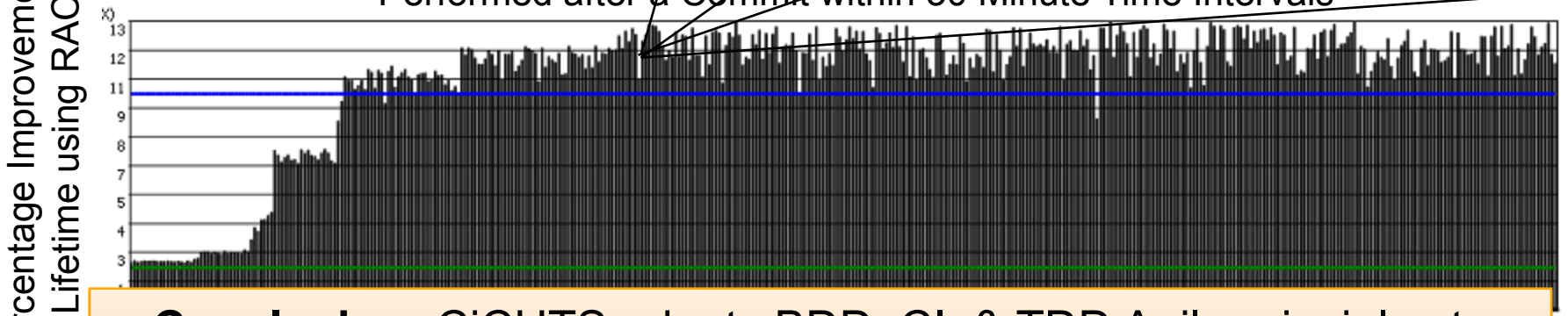


Results: Multiple Tests Performance

We are able to validate hypothesis 1 (H1) continuously throughout the development of RACE

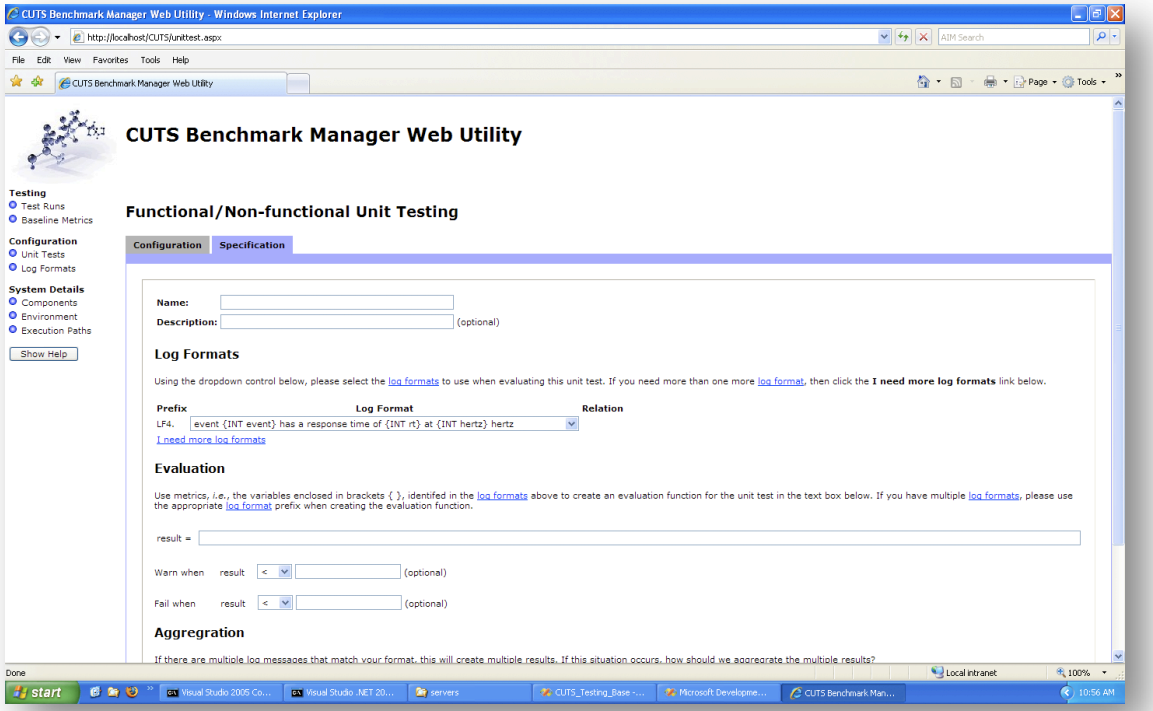


Continuous Testing of a
Performed after a Commit within 30 Minute Time Intervals

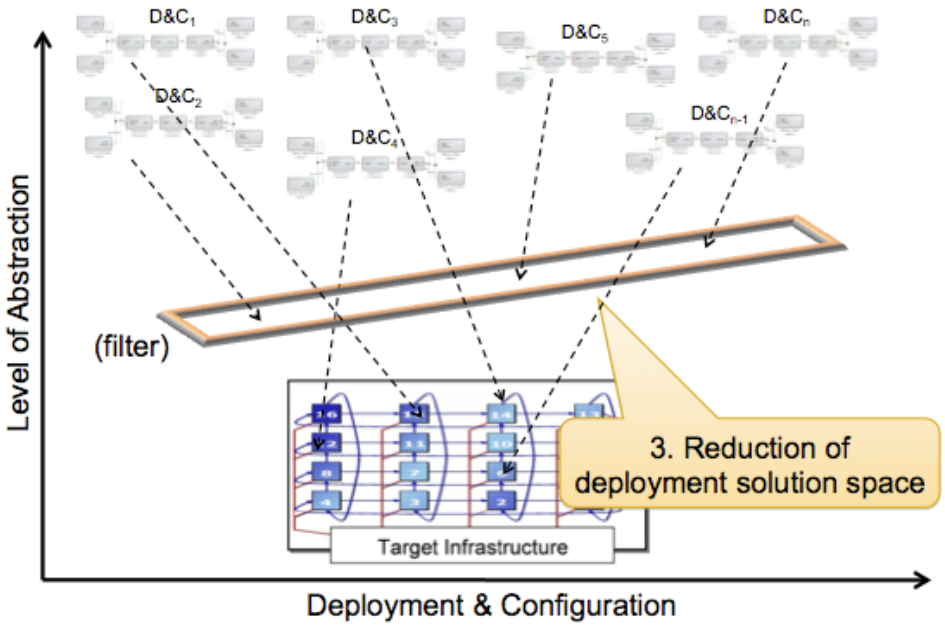


Conclusion: CiCUTS adapts BDD, CI, & TDD Agile principles to evaluate non-functional concerns (*e.g.*, QoS) of component-based DRE systems throughout the entire development phase

PART 6:
Demo using CiCUTS



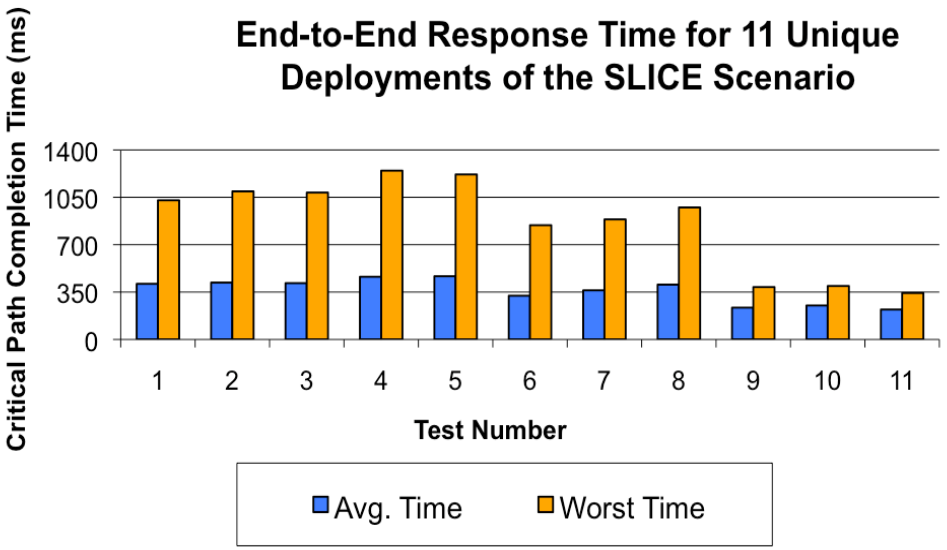
PART 7:
Deployment-based Analysis – Addressing Complexity #3

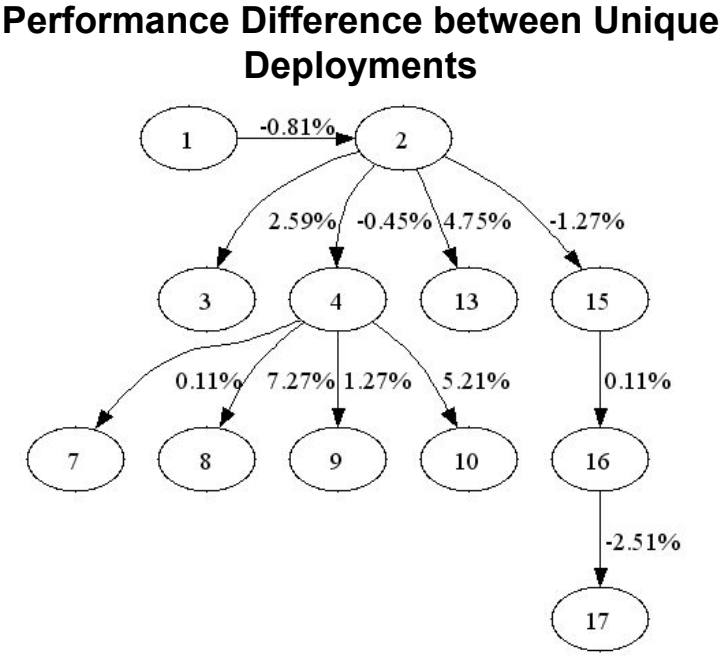


Deployment Solution Space Reduction

- The deployment solution space can be so enormous that locating ones that meet QoS requirements is like looking for a needle in a haystack
- What can be done to make this problem smaller?

Challenge: Evaluation of Unique Deployments



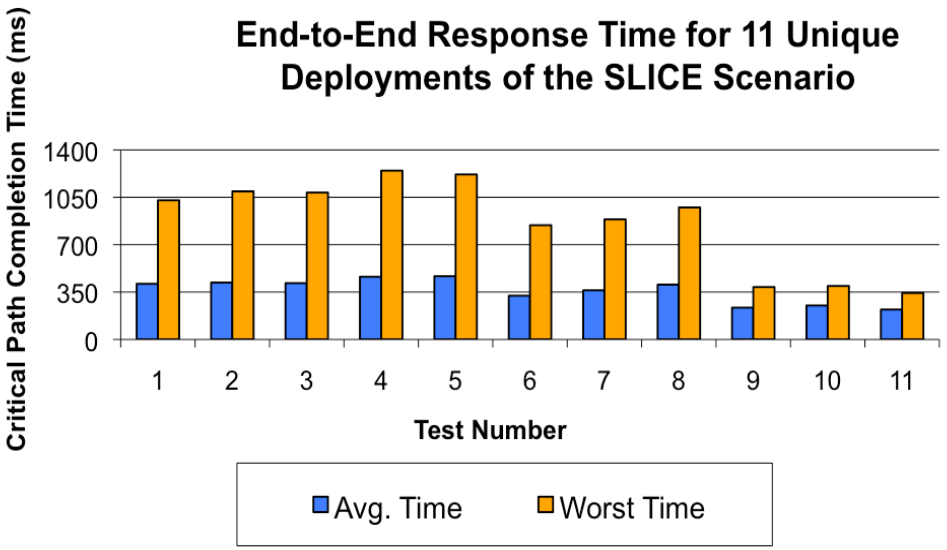


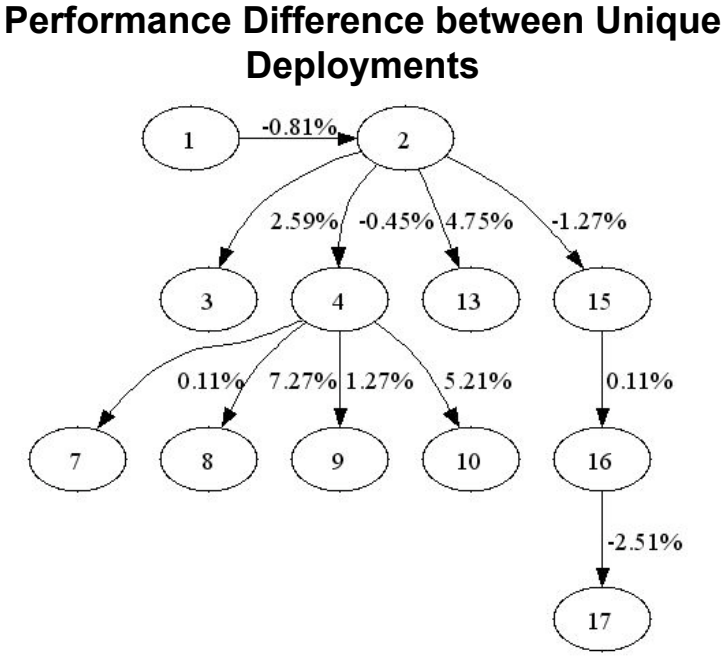
Ad-hoc Evaluation of Unique Deployments

Strategic Evaluation of Unique Deployments

- Different deployments are known to deliver different QoS, such as end-to-end worst execution time
- Evaluating each deployment can be time-consuming & costly

Challenge: Evaluation of Unique Deployments



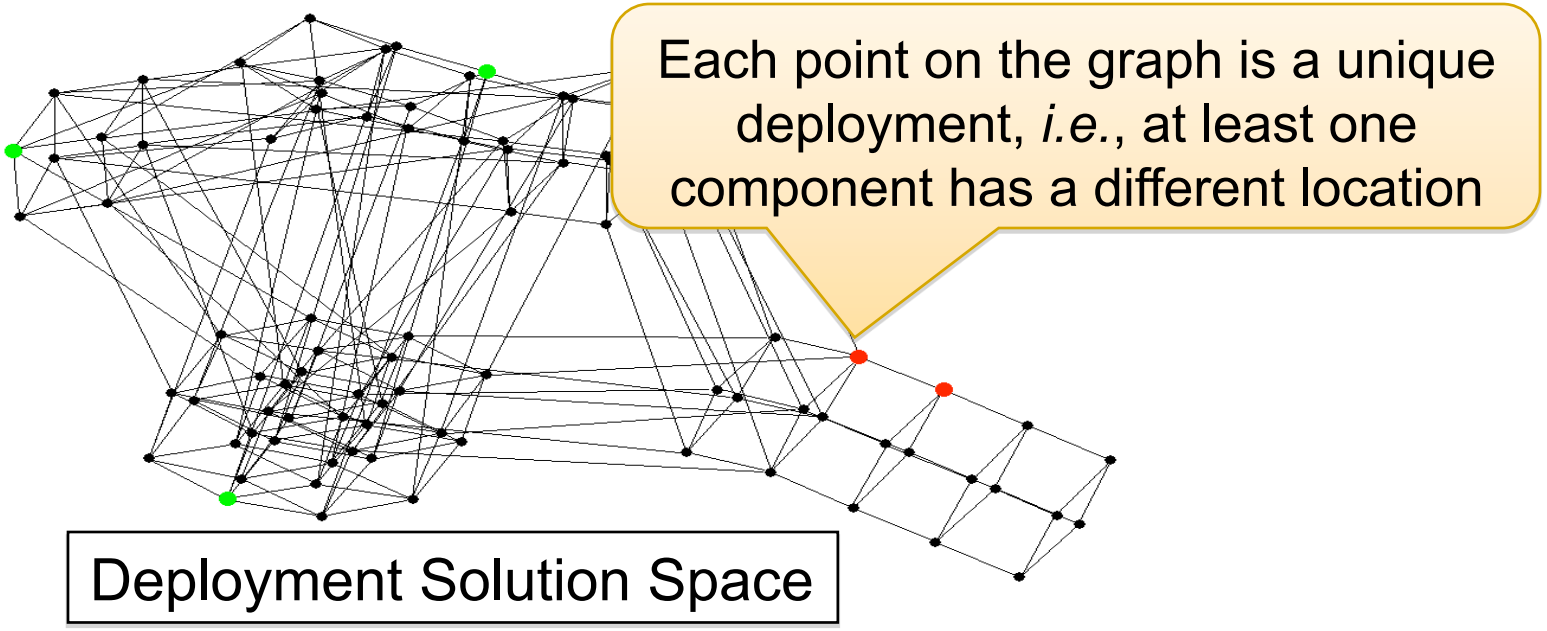


Ad-hoc Evaluation of Unique Deployments

Strategic Evaluation of Unique Deployments

- Different deployments are known to deliver different QoS, such as end-to-end worst execution time
- Evaluating each deployment can be time-consuming & costly

Challenge: Evaluation of Unique Deployments



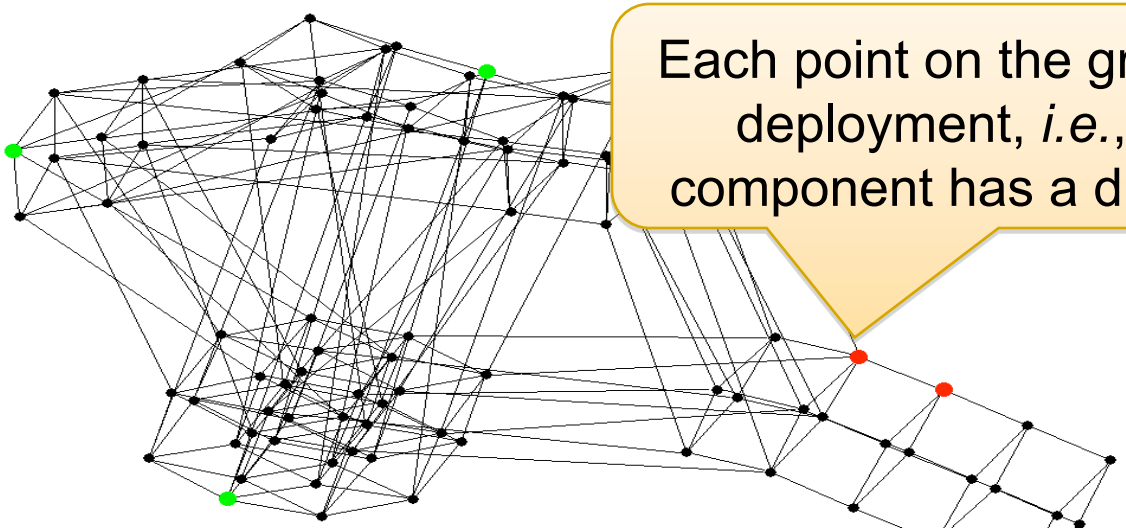
- Let *H* be the set of hosts in system S, C be the set of components in system *S, & D* be the set of deployments, then a *unique deployment* can be defined as:

$$\exists h \in H : C_{h,i} \neq C_{h,j};\ i,j \in D \wedge i \neq j$$

where $C_{h,I}$ is the set of components deployed on host *h* for deployment *i*

- A component-based system can have $|host|^{|components|}$ unique deployments in its solutions space

Challenge: Evaluation of Unique Deployments



- Let *H* be the set of hosts in system S, C be the set of components in system *S, & D* be the set of deployments, then a *unique deployment* can be defined as:

$$\exists h \in H : C_{h,i} \neq C_{h,j};\ i,j \in D \wedge i \neq j$$

where $C_{h,I}$ is the set of components deployed on host *h* for deployment *i*

- A component-based system can have $|\text{host}|^{|\text{components}|}$ unique deployments in its solutions space

In an agile development environment, evaluating *every* possible deployment is not feasible due to time constraints & constant change

PART 8:
Concluding Remarks

Recap of Technology Enablers

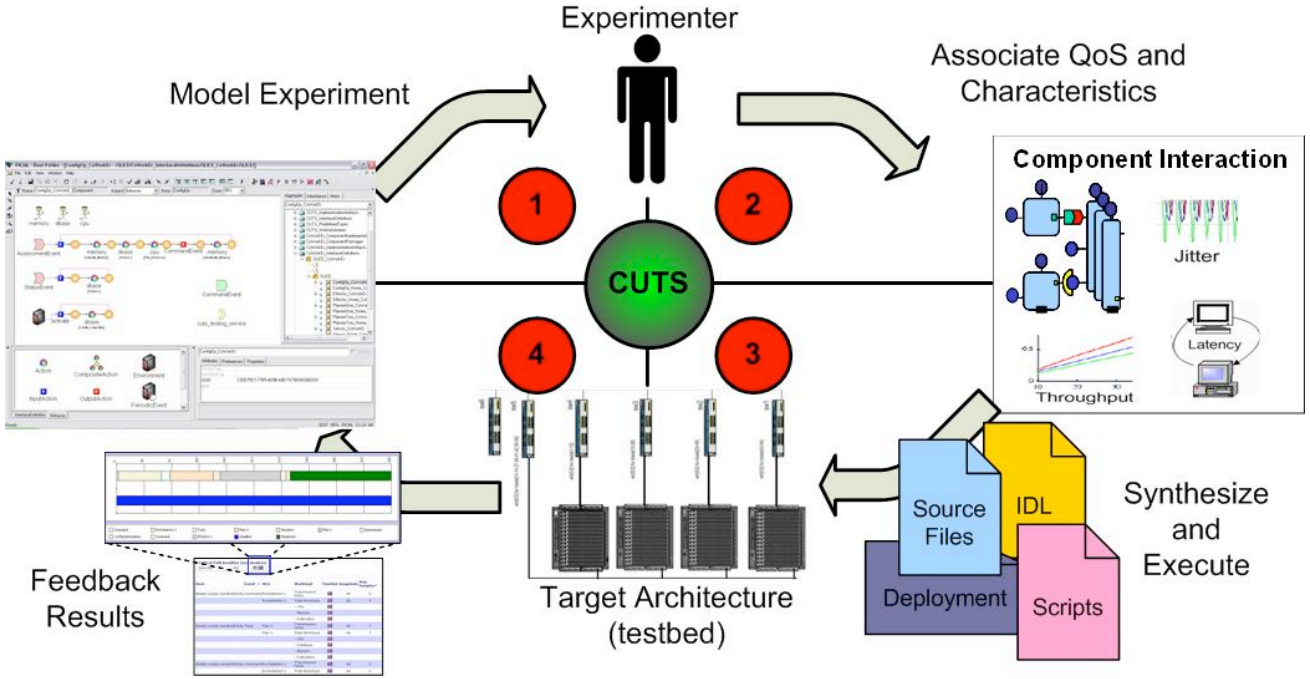
Reconciling Agile Software Development with Component -based Software Engineering for DRE Systems

Focus Area	Challenge	Approach	Enabler
Early Software Testing	• Test system auto-generation	• System execution modeling (SEM) Tools • DSMLs that require partial system knowledge	• CUTS
Continuous Software Integration	• Continuous testing & analysis of DRE systems	• Continuous integration + SEM • High-level performance specification & analysis	• CiCUTS

www.dre.vanderbilt.edu/CUTS

Conclusion

- System execution modeling (SEM) tools provided useful techniques & technologies that help adapt agile development techniques to concerns of distributed component-based systems
- There still remain many open issues to continue improving agile development of such systems



CUTS SEM tool is based on CoSMIC

CoSMIC is available from www.dre.vanderbilt.edu/cosmic

CUTS is available from www.dre.vanderbilt.edu/CUTS

PART 9:
Question & Answer Session