From PIMs to PSMs

Peter H Feiler, Dio DeNiz
Software Engineering Institute
Bruce Lewis
US Army
Chris Raistrick
Kennedy-Carter

Outline

MDA, xUML, and AADL

Domain Models and Bridges

xUML to AADL Translation

AADL Model Optimization

Important Abbreviations

see
omg.org/mda

The presentation describes:
A process for system development, known as
Model Driven Architecture (MDA)
which involves building
Platform-Independent Models (PIMs)
from which we derive
Platform-Specific Models (PSMs)
and/or
Platform-Specific Implementations (PSIs).

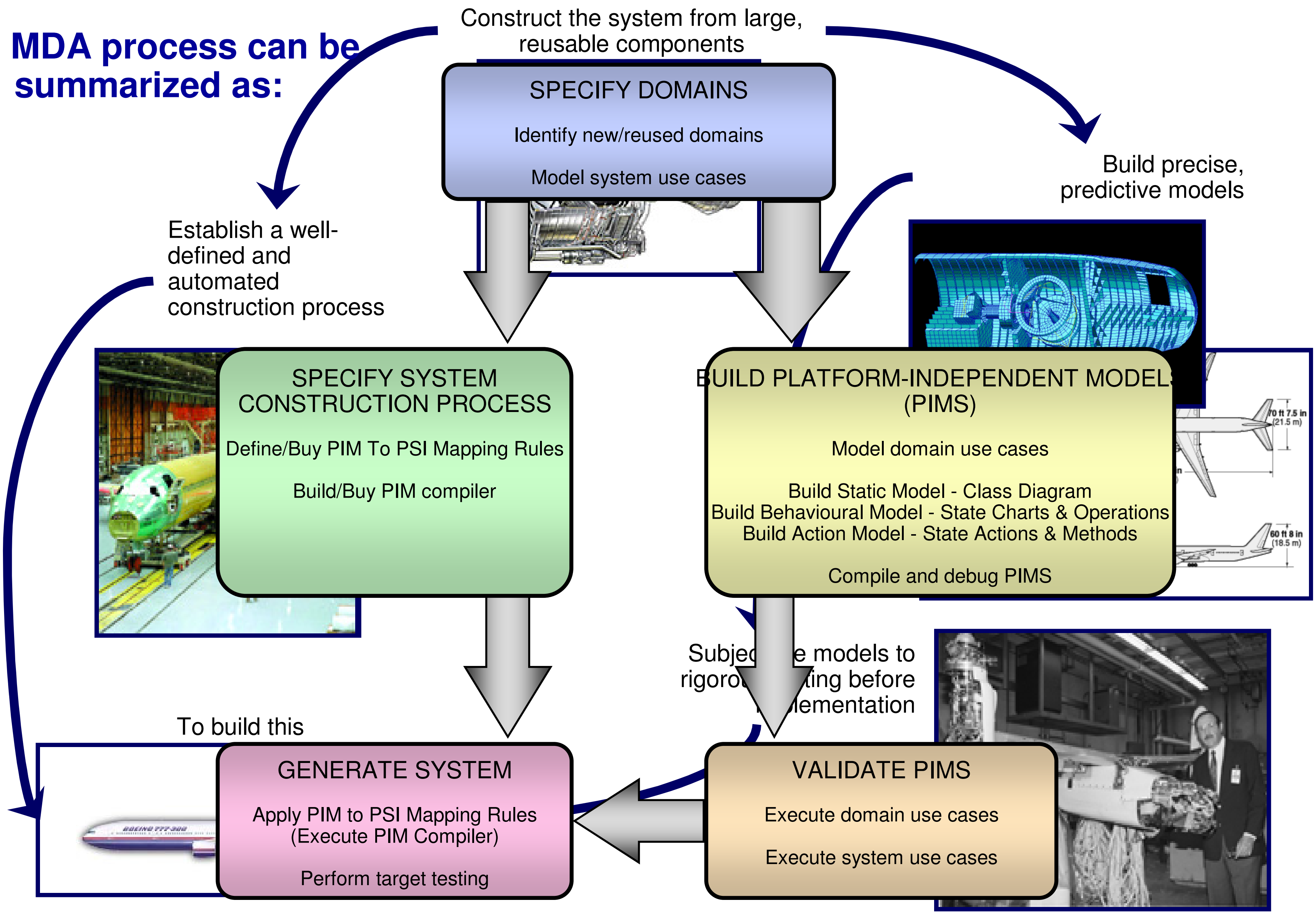
The models are represented using the notation known as the
Unified Modeling Language (UML).
Both the MDA process and the UML notation
are owned by the non-profit consortium known as the
Object Management Group (OMG).

Platform Independent Model

- A Platform Independent Model (PIM) is a technology agnostic model of some aspect of the system under study.
- A PIM contains no information about any of the following:
  - Hardware Architecture
  - Operating System
  - Programming Language
  - Database Technology
  - Internal Communication Technology
- It is therefore much simpler than a Platform-Specific Model (PSM)
- Use of Executable UML (xUML) allows construction of PIMs that are:
  - Precise
  - Complete
- PIMs built using xUML can be:
  - Executed to demonstrate compliance with functional requirements
  - Automatically translated into a complete Platform Specific Implementation using a suitable model translator
  - Used as executable specifications, forming the basis for contract-based procurement

Overview of the MDA Process

The MDA process can be summarized as:



Construct the system from large, reusable components

Build precise, predictive models

Establish a well-defined and automated construction process

Subjec e models to rigoro ting before lementation

To build this

SPECIFY DOMAINS

Identify new/reused domains

Model system use cases

SPECIFY SYSTEM CONSTRUCTION PROCESS

Define/Buy PIM To PSI Mapping Rules

Build/Buy PIM compiler

BUILD PLATFORM-INDEPENDENT MODELS (PIMS)

Model domain use cases

Build Static Model - Class Diagram
Build Behavioural Model - State Charts & Operations
Build Action Model - State Actions & Methods

Compile and debug PIMS

VALIDATE PIMS

Execute domain use cases

Execute system use cases

GENERATE SYSTEM

Apply PIM to PSI Mapping Rules
(Execute PIM Compiler)

Perform target testing

What is AADL?

- The SAE Architecture Analysis and Design Language (AADL) is an international standard for predictable model-based engineering of real- time and embedded computer systems.
- Intended fields of application are automotive systems, avionics and space applications, medical devices, and industrial process control equipment.
- The SAE AADL international standard consists of
  - a textual and graphical language with precise execution semantics for modeling the architecture of embedded software systems and their target platform;
  - a UML 2.0 profile for AADL that adds real-time embedded systems semantics of AADL to UML;
- AADL can be used to:
  - Represent embedded systems as component-based system architecture
  - Model component interactions as flows, service calls, and shared access
  - Model task execution and communication with precise timing semantics
  - Accommodate analyses such as reliability &safety- criticality through extensions

Ref: http://www.aadl.info/

xUML and AADL in the MDA Process

- xUML focuses on:
  - Service layers (domains)
  - Provided and Required services
  - Data structure (classes and attributes)
  - Processing (state machines and operations)
  - Interactions (signals and invocations)

- AADL focuses on:
  - Processors
  - Processes
  - Threads
  - Devices
  - Buses
  - Ports
  - Memory blocks

- xUML is strongly oriented towards Platform Independent Models…

  …that can be executed and analysed to assess functional capabilities…

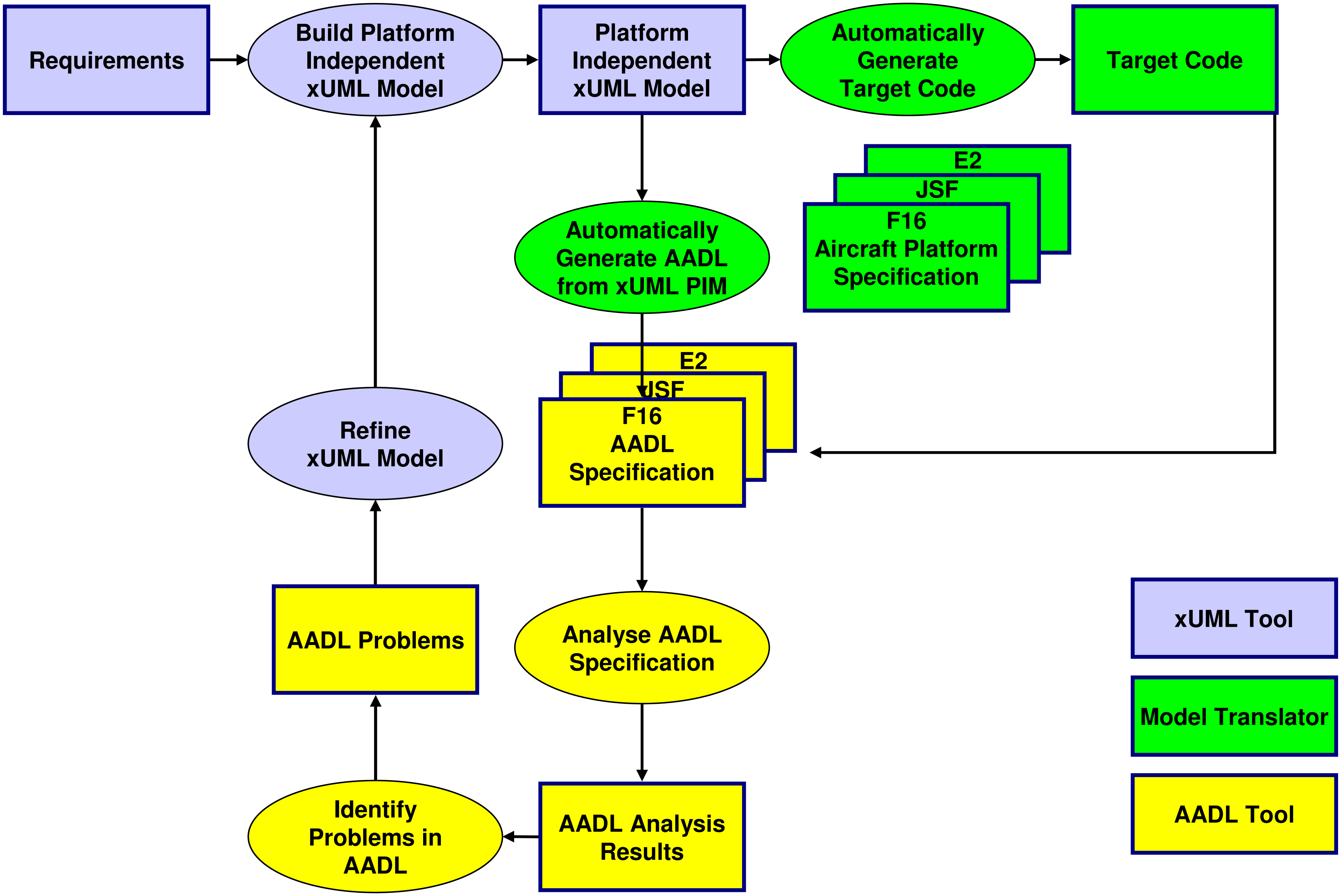
  …and used to generate several platform specific models, expressed using an appropriate language, for different aircraft types

- AADL is strongly oriented towards Platform Specific Models…

  …that can be executed and analysed to assess aircraft-specific performance characteristics…

  …and used as the basis for platform-specific implementations

An xUML-AADL Process



Requirements

Build Platform Independent xUML Model

Platform Independent xUML Model

Automatically Generate Target Code

Target Code

Automatically Generate AADL from xUML PIM

E2

JSF

F16

Aircraft Platform Specification

E2

JSF

F16

AADL Specification

Refine xUML Model

AADL Problems

Analyse AADL Specification

Identify Problems in AADL

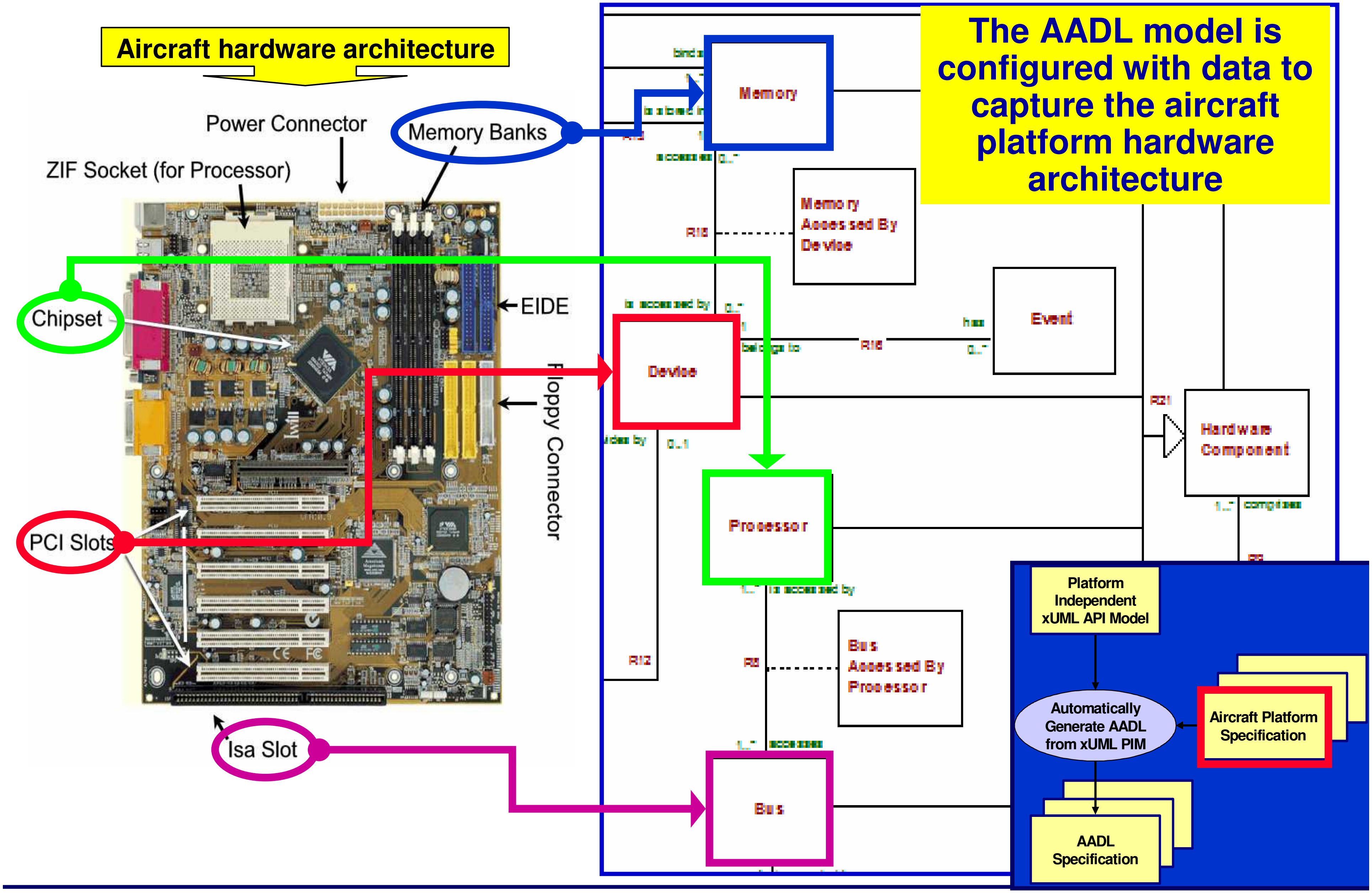
AADL Analysis Results

xUML Tool

Model Translator

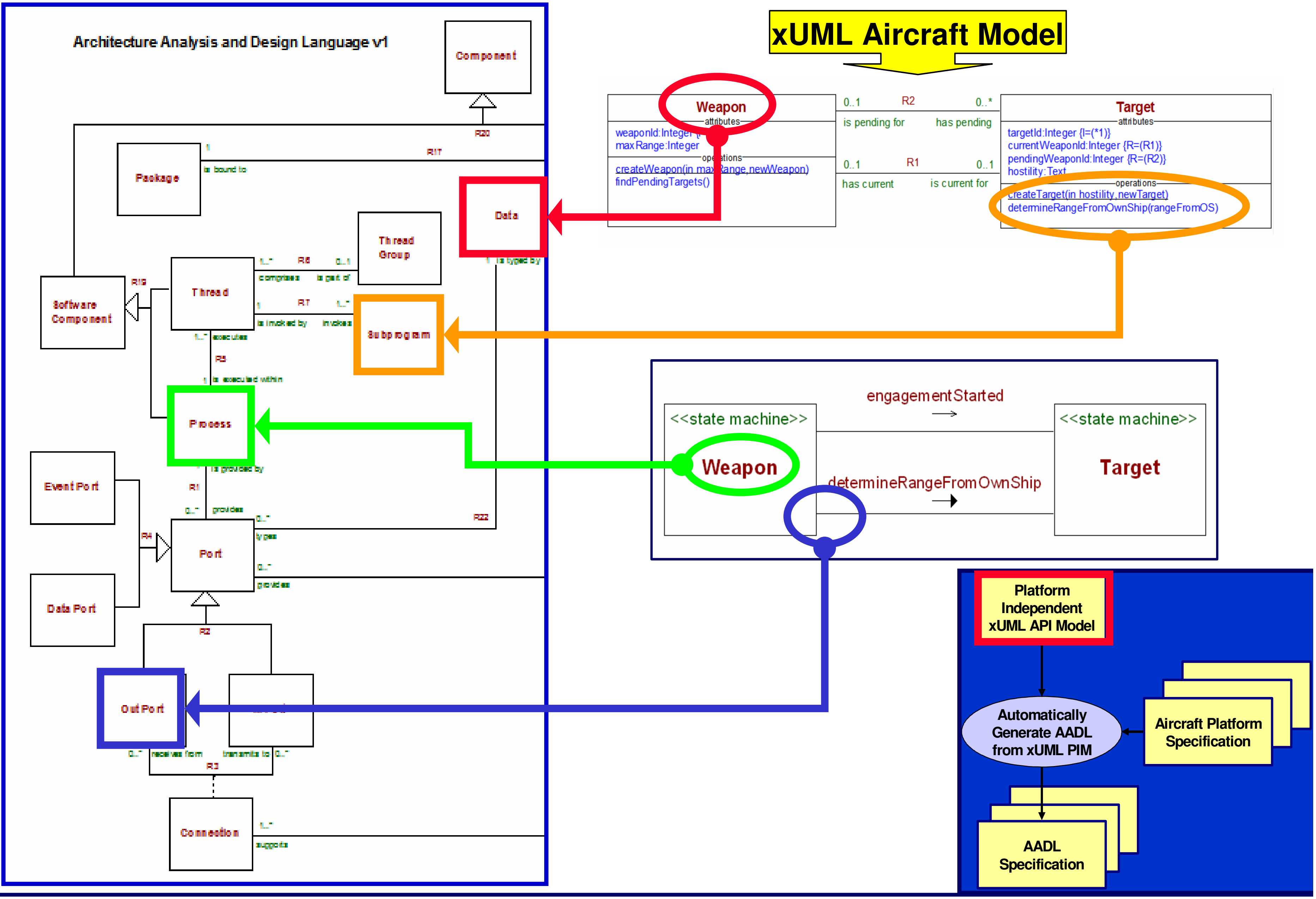
AADL Tool

Configure the AADL Model for the Hardware Components



The AADL model is configured with data to capture the aircraft platform hardware architecture

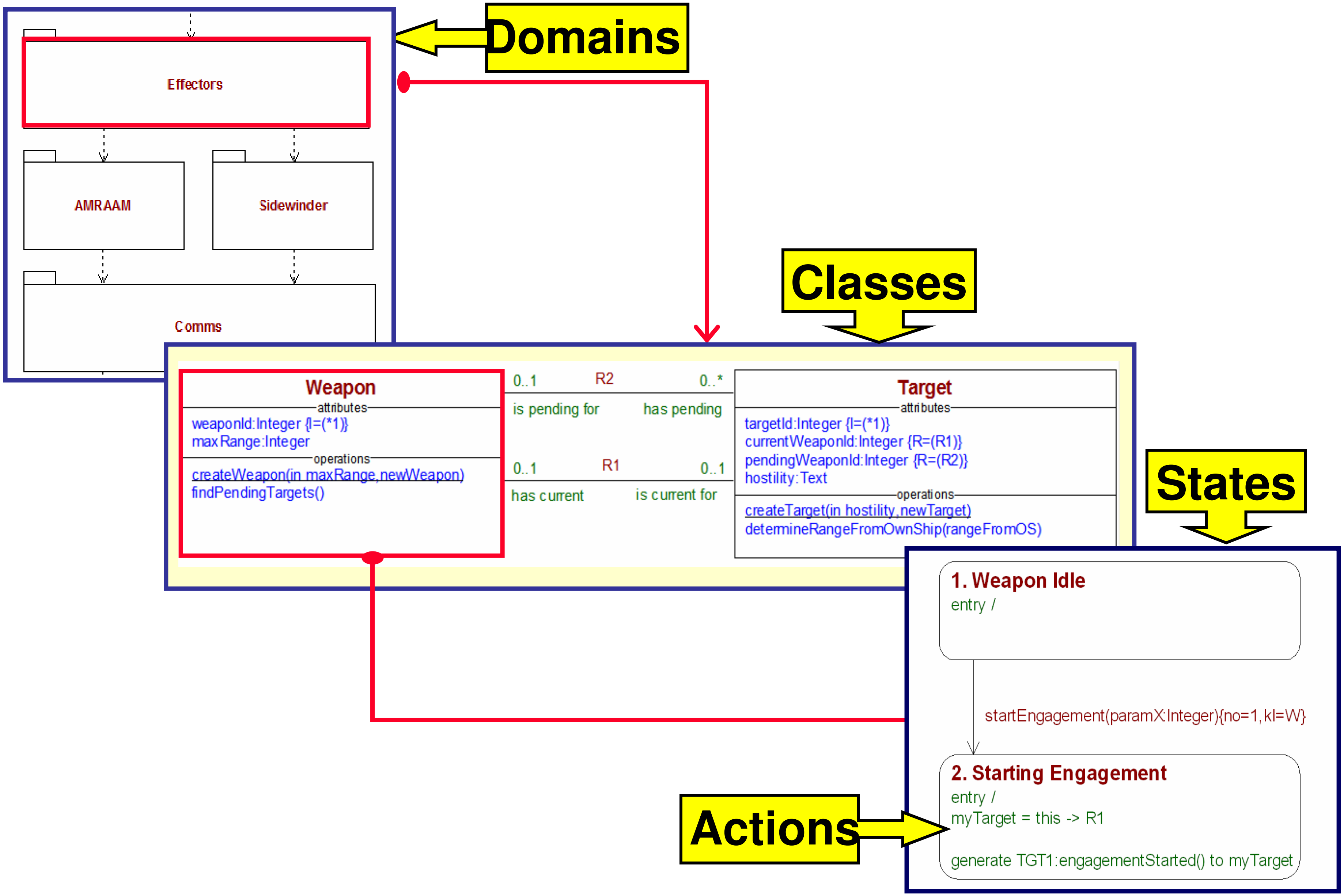
Populate the AADL Metamodel with Software Components



Outline

- MDA, xUML, and AADL
- Domain Models and Bridges
- xUML to AADL Translation
- AADL Model Optimization

The Primary xUML Models



Domains

Effectors

AMRAAM

Sidewinder

Comms

Classes

Weapon

attributes

weaponId:Integer {I=(*1)}
maxRange:Integer

operations

createWeapon(in maxRange,newWeapon)
findPendingTargets()

0..1 R2 0..*

is pending for — has pending

0..1 R1 0..1

has current — is current for

Target

attributes

targetId:Integer {I=(*1)}
currentWeaponId:Integer {R=(R1)}
pendingWeaponId:Integer {R=(R2)}
hostility:Text

operations

createTarget(in hostility,newTarget)
determineRangeFromOwnShip(rangeFromOS)

States

1. Weapon Idle

entry /

startEngagement(paramX:Integer){no=1,kl=W}

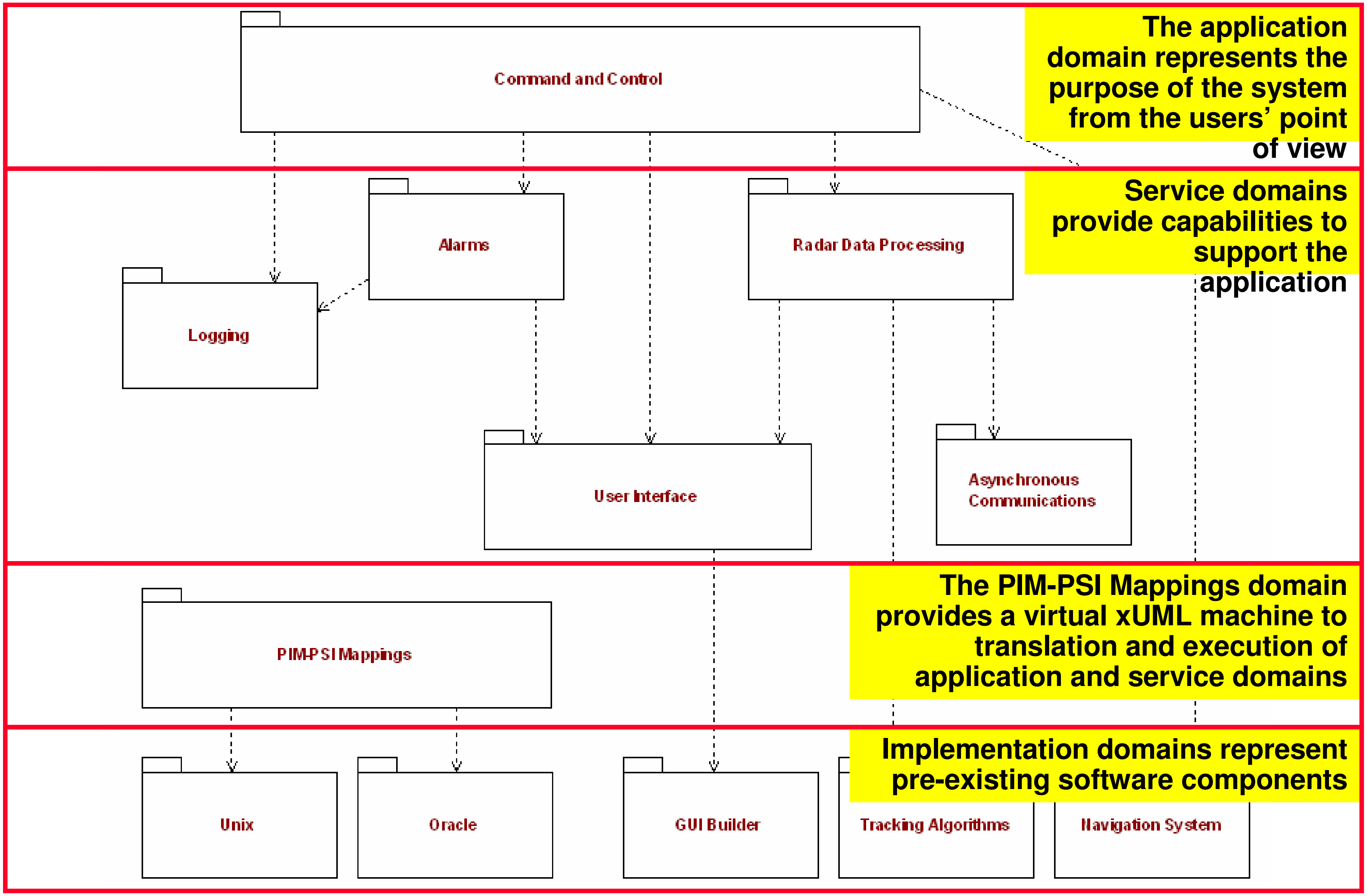
2. Starting Engagement

entry /
myTarget = this -> R1

generate TGT1:engagementStarted() to myTarget

Actions

The Four Domain Layers



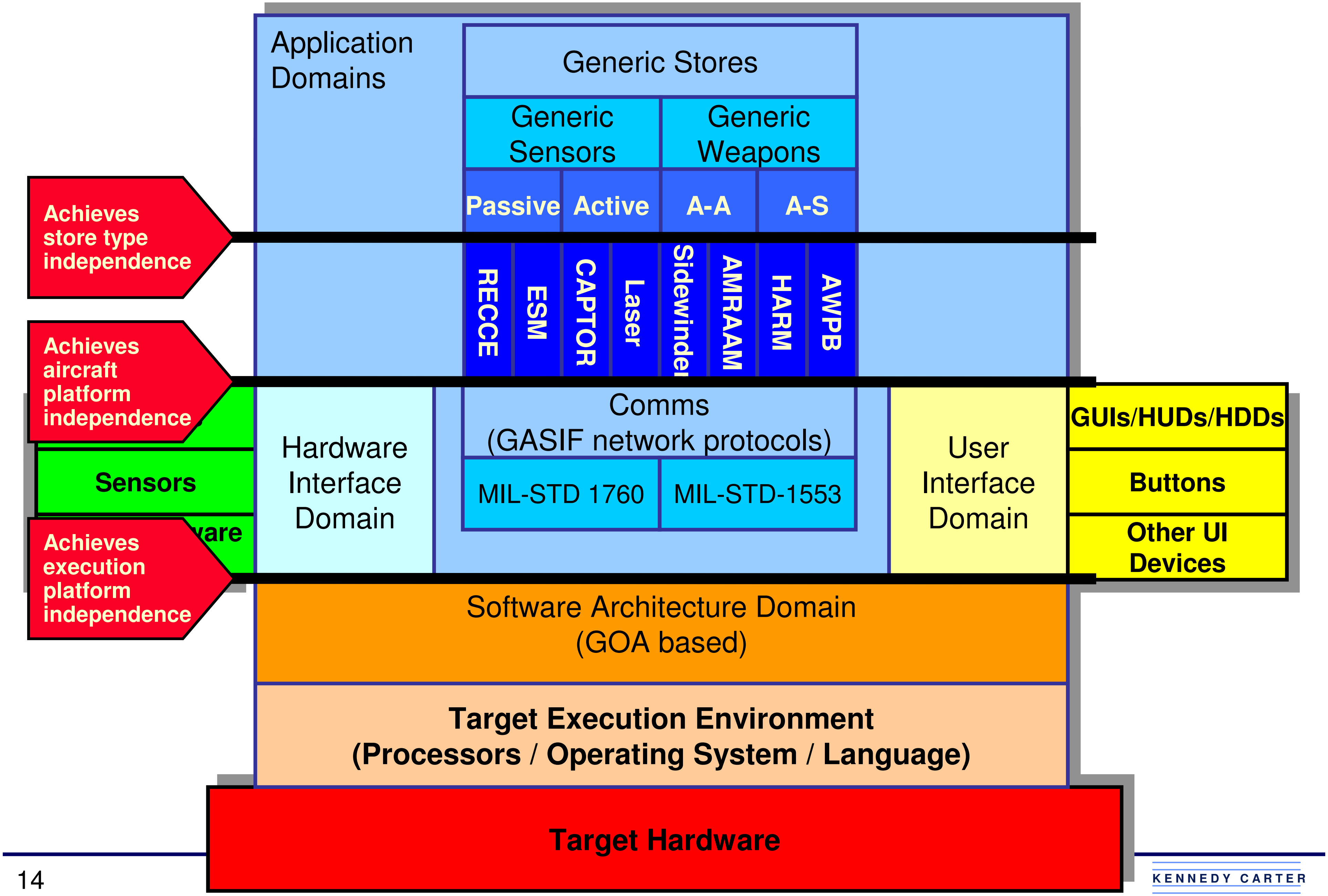
The application domain represents the purpose of the system from the users’ point of view

Service domains provide capabilities to support the application

The PIM-PSI Mappings domain provides a virtual xUML machine to translation and execution of application and service domains

Implementation domains represent pre-existing software components

Use Domains to Isolate Areas of Change (or Platforms)



Application Domains

Generic Stores

Generic Sensors

Generic Weapons

Passive **Active** **A-A** **A-S**

RECCE
ESM
CAPTOR
Laser
Sidewinder
AMRAAM
HARM
AWPB

Achieves store type independence

Achieves aircraft platform independence

Achieves execution platform independence

Sensors

…ware

Hardware Interface Domain

Comms
(GASIF network protocols)

MIL-STD 1760

MIL-STD-1553

User Interface Domain

GUIs/HUDs/HDDs

Buttons

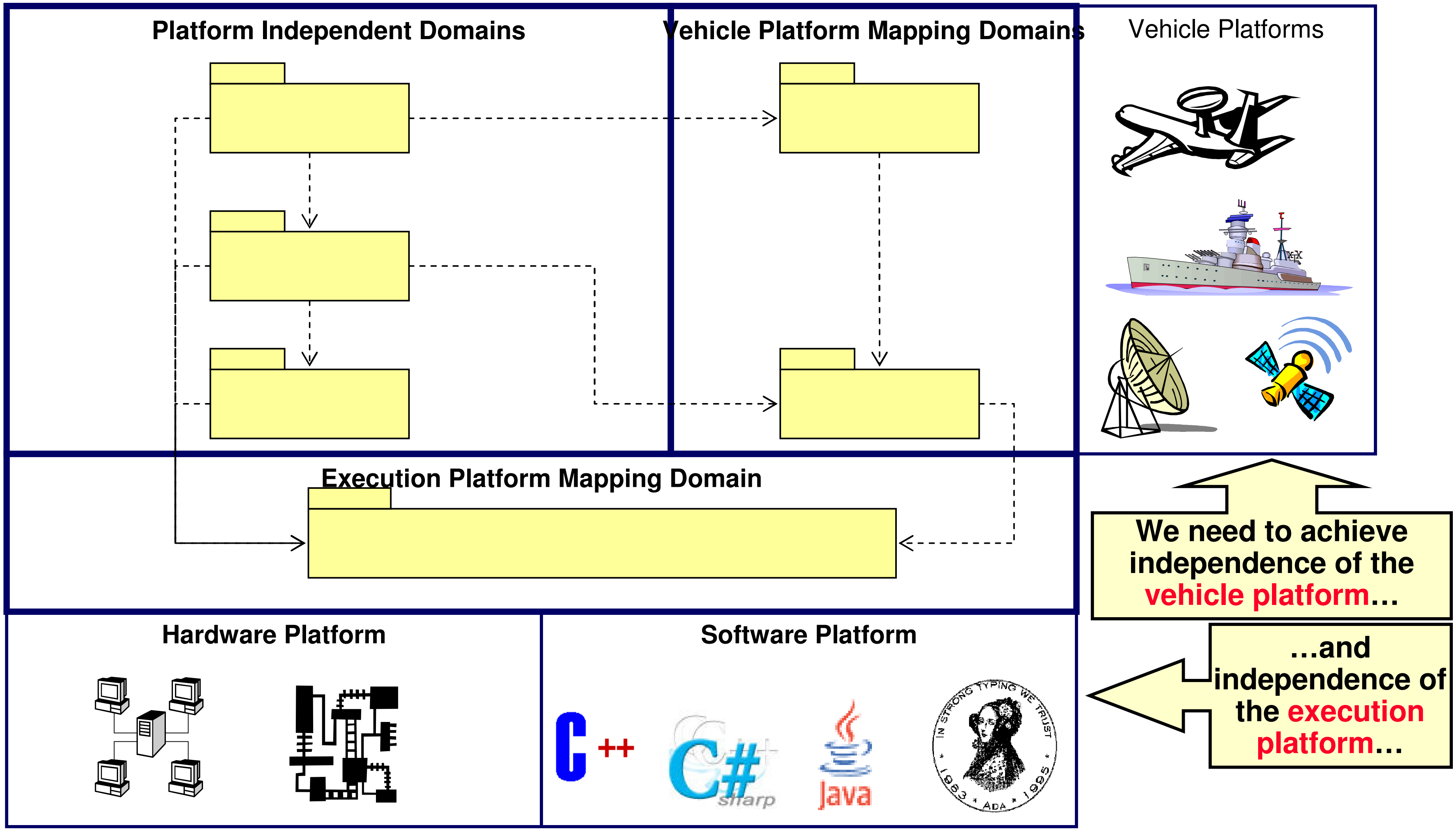
Other UI Devices

Software Architecture Domain
(GOA based)

Target Execution Environment
(Processors / Operating System / Language)

Target Hardware

Domain Architecture for Platform Independence

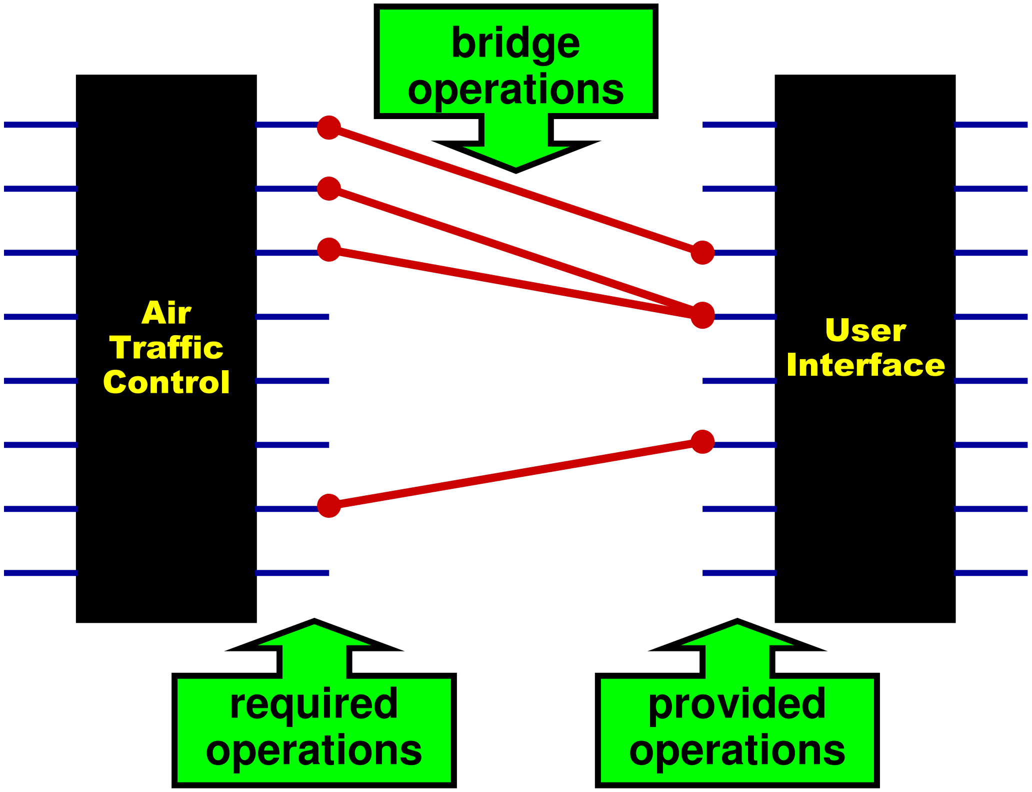


We need to achieve independence of the **vehicle platform…**

…and independence of the **execution platform…**

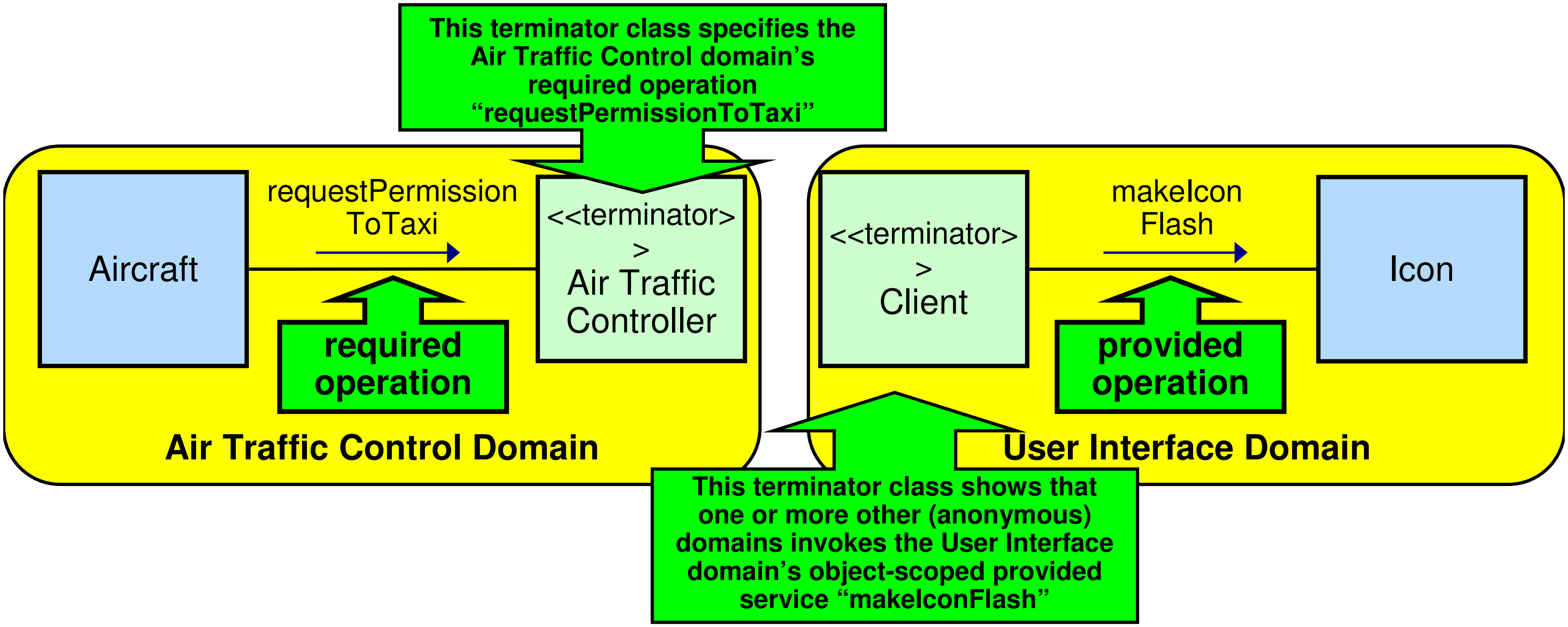
Elements of a Domain’s Interfaces

Each domain can be thought of as an “integrated circuit” of classes (the black box)…
…with a set of provided and required operations (the pins)…
…that can be connected together into a system (the wiring)



The Terminator Classes Provide the Placeholders…

The <<terminator>> classes represent placeholders for the required operations for the domain to which they belong…
…and show which provided operations are invoked by other domains

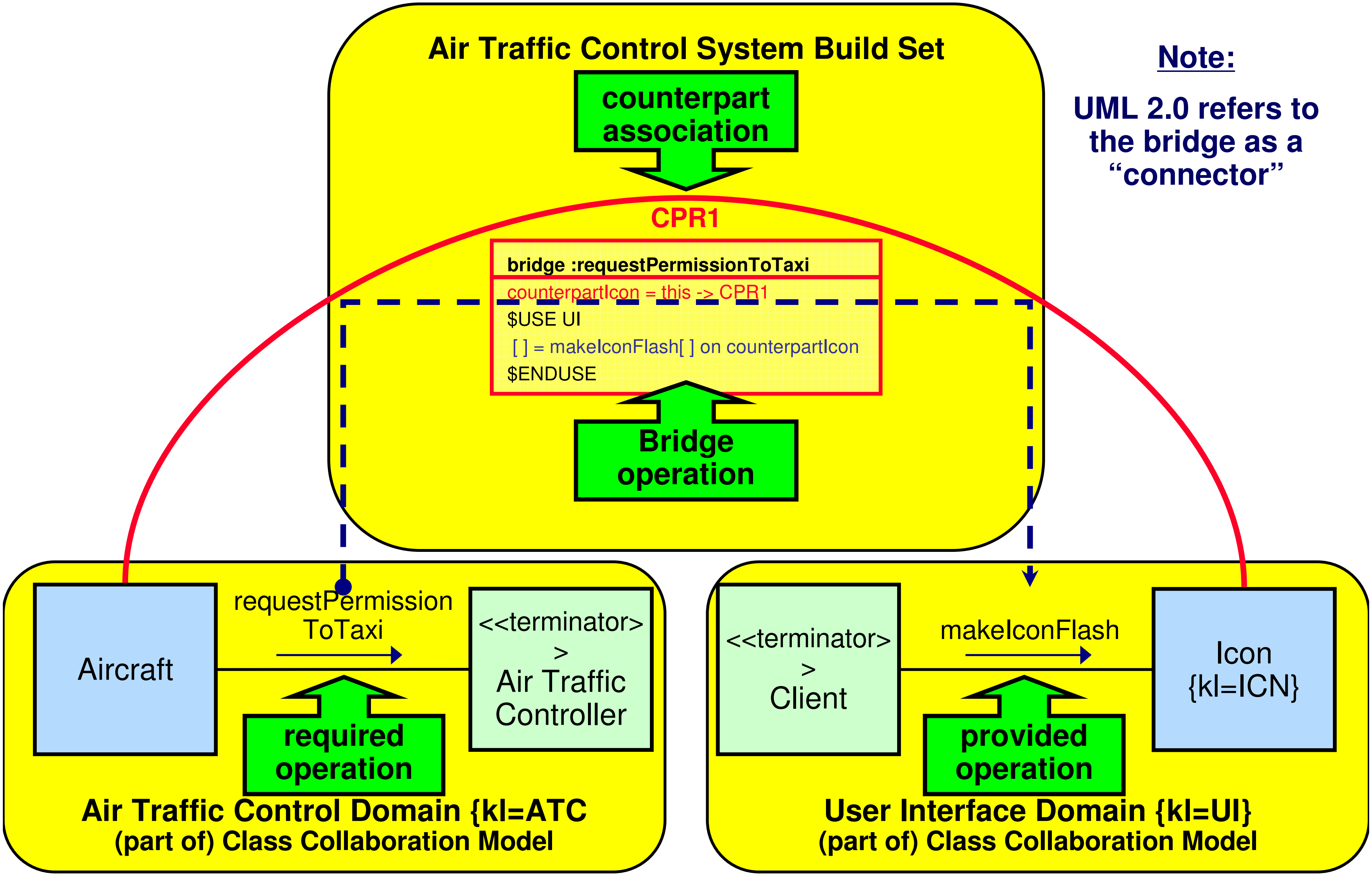


This terminator class specifies the Air Traffic Control domain's required operation "requestPermissionToTaxi"

This terminator class shows that one or more other (anonymous) domains invokes the User Interface domain's object-scoped provided service "makeIconFlash"

The <<terminator>> classes are constrained such that they can only have operations. They cannot have attributes, associations, methods, state machine.

The “Wiring” Is Specified in a Build Set…



Note:

UML 2.0 refers to the bridge as a “connector”

Outline

MDA, xUML, and AADL

Domain Models and Bridges

xUML to AADL Translation

AADL Model Optimization

Purpose of Translation

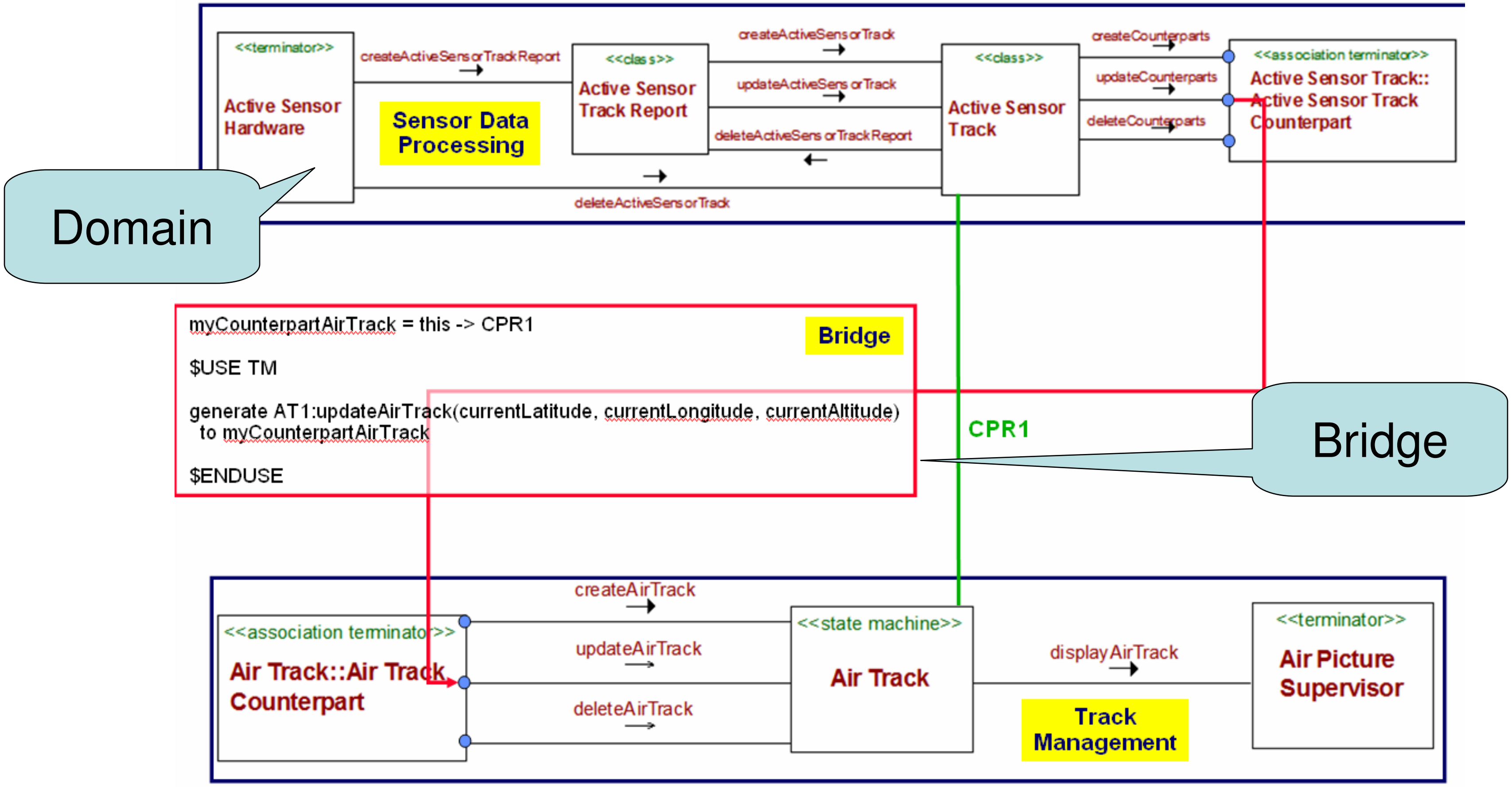
Analyze the runtime characteristics of a model expressed in xUML

Improve runtime structure

Options:

- Active object == thread (logical thread) & thread optimization to OS threads
- Define task architecture (OS threads) & active object -> thread mapping

Sample



```
myCounterpartAirTrack = this -> CPR1

$USE TM

generate AT1:updateAirTrack(currentLatitude, currentLongitude, currentAltitude)
  to myCounterpartAirTrack

$ENDUSE
```

Mapping Domains

Domains are mapped to packages in AADL

Every definition in the public section

package xUMLBasicTypes

public

...

end xUMLBasicTypes;

package SensorDataProcessingDomain

public

...

end SensorDataProcessingDomain;

package TrackManagementDomain

public

...

end TrackManagementDomain;

Finding xUML Threads

Two Sources

- External Device Stimuli
- State Machines

In Our Example

- Active Sensor Hardware
- AirTrack State Machine

Translation of Message Semantics

xUML Semantics

- Closed Blocking. This represents a function call where the caller can send data and expects and waits for an answer from the callee before continuing its execution.
- Closed Non-Blocking. In this case the caller also expects an answer but it will not wait to get it before continuing its execution. Instead it queries for the answer at a later time.
- Open. This involves a transfer of data from the caller to the callee. The caller does not wait for the completion of the callee neither expects any answer from it.

AADL Semantics

- Closed Blocking. In this case the callee is a subprogram and the message from the caller to the callee a subprogram call.
- Closed Non-Blocking. In this case the callee is a thread (and hence the caller is another thread). The message is a data port connection from caller to callee and an event port connection from the callee to the caller to notify the completion of the execution.
- Open. In this case the caller and the callee are both thread and the message is only a event data port connection from the caller to the callee

AADL Threads for Example

```
thread ActiveSensorThread
 features
  createActiveSensorTrackReport: in event data port CreateActiveSensorTrackEvent;
  initializeAirTrack: out event data port
TrackManagementDomain::InitializeAirTrackEvent;
  updateAirTrack: out event data port TrackManagementDomain::UpdateAirTrackEvent;
  deleteAirTrack: out event data port TrackManagementDomain::DeleteAirTrackEvent;
end ActiveSensorThread;
thread AirTrackThread
 features
  initializeAirTrack: in event data port InitializeAirTrackEvent;
  updateAirTrack: in event data port UpdateAirTrackEvent;
  deleteAirTrack: in event data port DeleteAirTrackEvent;
end AirTrackThread;
```

Implicit Object Management

Objects are assumed to be managed by its class in xUML

- Find objects
- Manage object memory for creation/deletion

Need to be explicit in AADL

- In the form of “Collection”

Sample Collection

```
data ActiveSensorTrackReport
 features
  initialize: subprogram InitializeActiveSensorTrackReportInstance;
  update: subprogram UpdateActiveSensorTrackReportInstance;
  delete: subprogram DeleteActiveSensorTrackReportInstance;
end ActiveSensorTrackReport;
data ActiveSensorTrackReportCollection
 features
  find :  subprogram FindActiveSensorTrackReportCollection;
  create: subprogram CreateActiveSensorTrackReportCollection;
  delete: subprogram DeleteActiveSensorTrackReportCollection;
  update: subprogram UpdateActiveSensorTrackReportCollection;
end ActiveSensorTrackReportCollection;
```

Call Sequences

```
thread implementation ActiveSensorThread.Impl
 subcomponents
  reportCollection: data ActiveSensorTrackReportCollection;
  activeSensorTrackCollection: data ActiveSensorTrackCollection;
  airTrackCollection: data AirTrackCollection;
 calls
  createReport: { find1: subprogram ActiveSensorTrackReportCollection.find;
              create1: subprogram ActiveSensorTrackReportCollection.create;};
  updateReport: { find2: subprogram ActiveSensorTrackReportCollection.find;
                                  update1: subprogram ActiveSensorTrackReportCollection.update;};
  deleteReport: { find3: subprogram ActiveSensorTrackReportCollection.find;
                                  delete1: subprogram ActiveSensorTrackReportCollection.delete;};
 connections
  c1: event data port create1.initializeAirTrack->initializeAirTrack;
  c2: event data port update1.updateAirTrack->updateAirTrack;
  c3: event data port delete1.deleteAirTrack->deleteAirTrack;
  p1: parameter createActiveSensorTrackReport->find1.report;
  p2: parameter createActiveSensorTrackReport->create1.report;
  p3: parameter createActiveSensorTrackReport->find2.report;
  p4: parameter createActiveSensorTrackReport->update1.report;
  p5: parameter createActiveSensorTrackReport->find3.report;
  p6: parameter createActiveSensorTrackReport->delete1.report;
end ActiveSensorThread.Impl;
```

Final System

```
process TrackingProcess
 features
  createActiveSensorTrackReport : in event data port
SensorDataProcessingDomain::CreateActiveSensorTrackEvent;
end TrackingProcess;
process implementation TrackingProcess.Impl
 subcomponents
  sensorThread: thread
SensorDataProcessingDomain::ActiveSensorThread;
  airTrackThread: thread TrackManagementDomain::AirTrackThread
{xUML::Multiplicity => 100;};
 connections
  c1: event data port sensorThread.initializeAirTrack-
>airTrackThread.initializeAirTrack {xUML::Connection_Multiplicity =>
OneToOne;};
  c2: event data port sensorThread.updateAirTrack-
>airTrackThread.updateAirTrack {xUML::Connection_Multiplicity =>
OneToOne;};
  c3: event data port sensorThread.deleteAirTrack-
>airTrackThread.deleteAirTrack {xUML::Connection_Multiplicity =>
OneToOne;};
  c4: event data port createActiveSensorTrackReport-
>sensorThread.createActiveSensorTrackReport;
end TrackingProcess.Impl;
```

```
device ActiveSensorDevice
 features
  createActiveSensorTrackReport: out event data port
SensorDataProcessingDomain::CreateActiveSensorTrackEvent;
end ActiveSensorDevice;
processor MyProcessor
end MyProcessor;
system Final
end Final;
system implementation Final.Impl
 subcomponents
  sensor: device ActiveSensorDevice;
  proc: processor MyProcessor;
  trackProcess: process TrackingProcess;
 connections
  c1: event data port sensor.createActiveSensorTrackReport-
>trackProcess.createActiveSensorTrackReport;
end Final.Impl;
```

Performance Analysis

Properties to be added

- End-to-end latency requirements
- Periodicity of events, both external (e.g. sensor interrupts) and internal (timers – could be extracted from the xUML model)
- Execution time of subprograms
- Processor Speed
- Network Speed

Outline

MDA, xUML, and AADL

Domain Models and Bridges

xUML to AADL Translation

AADL Model Optimization

Mapping into Operating System Threads

Transformation of logical thread model

- Threads in transformed model represent OS threads
- Logical threads become subprogram calls in OS thread

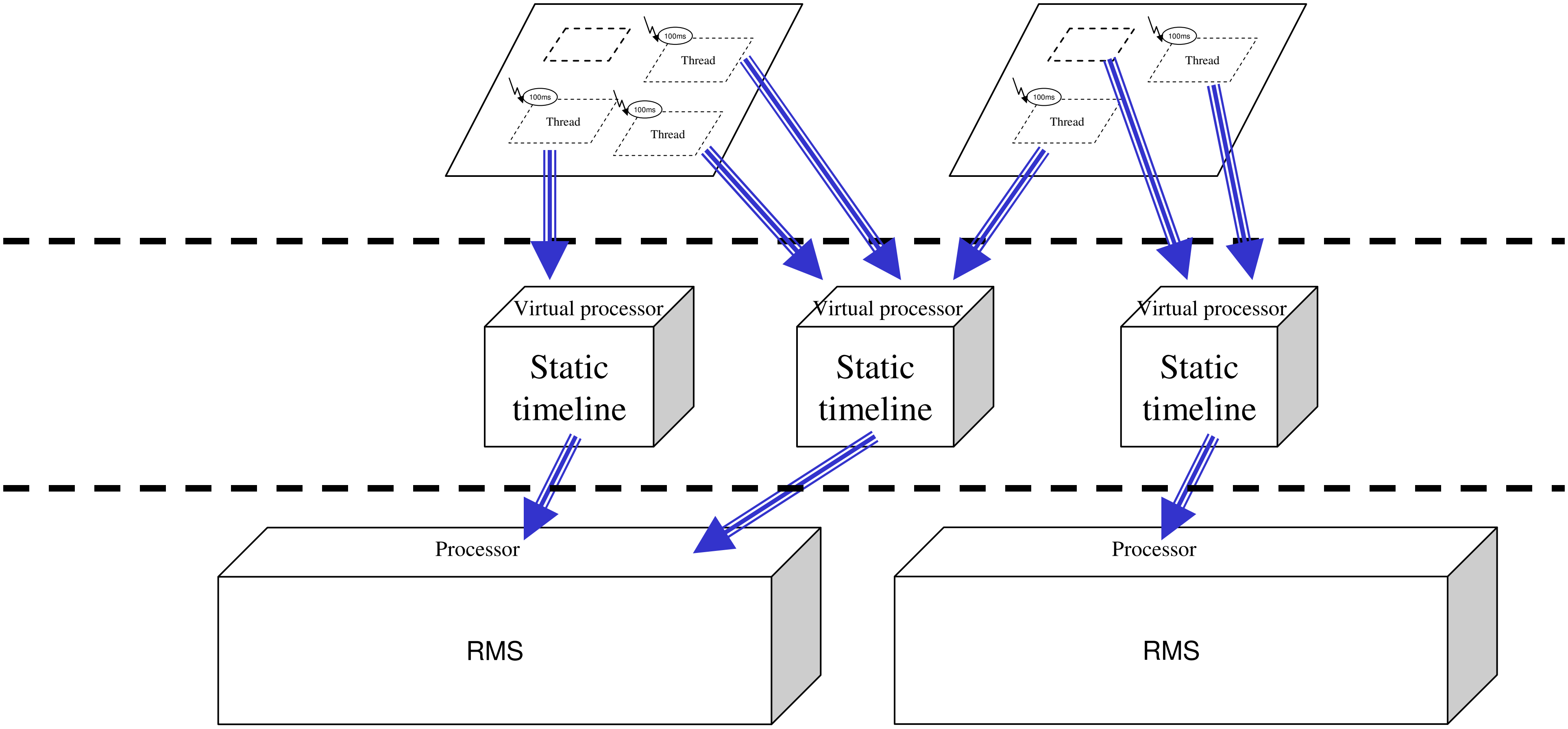
Thread groups to represent thread mappings

- Assignment by containment grouping
- Rate group optimization

Virtual processor to represent OS thread

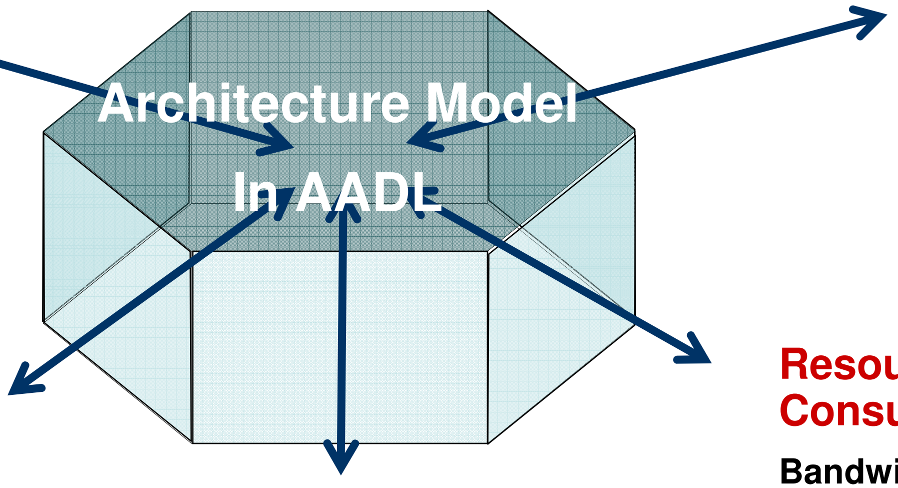
- Virtual processor as hierarchical scheduler
- Logical thread binding to virtual processor
- Virtual processor binding or containment

Two-Level Thread Binding



Predictive Analysis Across Perspectives

Single-Source Model Reduces Cost of Model Evolution



Availability & Reliability

MTBF

FMEA

Hazard analysis

Security

Intrusion

Integrity

Confidentiality

Data Quality

Data precision/ accuracy

Temporal correctness

Confidence

Real-time Performance

Execution time/ Deadline

Deadlock/starvation

Latency

Resource Consumption

Bandwidth

CPU time

Power consumption

Observations

PSMs require more than UML offers

- AADL is targetedat runtime architecture
- OMG MARTE compatible with AADL

Mapping xUML design patterns

- Active objects, terminators, and bridges
- Functional interface
- connection semantics in bridge patterns

AADL-based runtime architecture model

- Logical thread and OS threads
- Basis of multi-dimensional multi-fidelity analysis of operational properties
- Generation of application specific runtime system implementation

Questions?

Contact info:

Peter H. Feiler

phf@sei.cmu.edu

412-268-7790