Three-Dimensional Components and How to Achieve True Platform Independence

Bran Selic
President, Malina Software Corp.
CANADA
Software systems whose functional correctness critically depends on meeting one or more quality of service (QoS) requirements

- Performance, throughput, energy consumption, reliability, etc.
- Usually real-time embedded systems but not exclusively (e.g., financial trading systems, gaming/simulation systems)

Some key design and development challenges of QoS-constrained systems:

- Coping with the complexity of the physical world
- Meeting QoS requirements cost effectively in the face of technological and resource limitations
Overview

- The 3D Component Model
- On Software Platforms
- The MARTE UML Profile Standard
- The CERAS MDD Research Initiative
The Classical Software Component Model

- “Black box” model
  - Fully hides its implementation
    - Separation of concerns
    - Flexibility
  - All interactions occur through its interface
- Basic Postulate: *To understand the capabilities of a black-box component it is sufficient to know its interface*
Improvements to the Naive Component Model

Separation of interface *purpose* from interface *type*

Port 1 provides InterfaceType1

Port 2 provides InterfaceType2

Port 3 provides InterfaceType1

Multiple interface *instances* to distinguish between different (kinds of) collaborators

Bidirectional interfaces (interfaces as two-way contracts between peers)

Dynamic interface types (beyond just signatures)

However, we need to refine this component model yet more!
Next Refinement: Adding Quality of Service (QoS)

- For many (most?) applications, system “performance” (i.e., quality of service) is a fundamental aspect of correctness.

![Diagram of a reusable component providing interface type 1 with performance metrics]

- Worst-case response time ≤ 5 ms
- Service availability
- Availability ≥ 99.999%
Basic Component Compositional Patterns

Peer-to-peer Connection

Containment

Layering

What is the nature of this interaction?
Analyzing the Layering Pattern

• Upper layers are existentially dependent on lower layer
  ▪ But upper layer does not encapsulate the lower layers ⇒ different from composition

• Lower layer is independent of the upper layer entities, e.g.:

  ![Diagram showing layering pattern]

  - The lower layer provides a set of shared implementation services
    - These cannot be encapsulated because they may be shared by more than one component
In complex systems, layering is a complex multidimensional relationship

- e.g., 7-layer model of Open System Interconnection (OSI)
Layering implies differentiating two kinds of component interfaces

- Implementation-independent *peer* interfaces
- Implementation-specific *layer* interfaces (*service access points*)

![Diagram of component framework]

**Component Framework**
- File System
- IPC Service
- Timing Service

**Peer interface**

**Layer interface (SAP)**
“To understand the capabilities of a black-box component it is sufficient to know its interface”

- Does this mean we have to know its layer interfaces as well?
- Why should we care about them? Aren’t they implementation specific?
Overview

- The 3D Component Model
- On Software Platforms
- The MARTE UML Profile Standard
- The CERAS MDD Research Initiative
The Impact of Construction Materials in Engineering

Grass hut

Construction materials (and tools) can have a fundamental impact on design in traditional engineering.

Early 20th century skyscraper
Traditional Engineering Design

Functional Requirements

Construction Materials

Quantitative Requirements

Design

\[ \Xi = \cos(\eta + \pi/2) + \xi*5 \]
The concerns are serialized with the functional ones given significantly greater priority.

Considerations of potential impact of technological characteristics on design are often ignored and even actively discouraged.
Rationale

- **Technology-independent design**
  - Ensures portability to different configurations and technologies
  - Simplifies design problem by separating concerns – allows proper focus on application itself

- **“Platform independent” design approach**
  - Disregard irrelevant technological detail during design
Yet...Could Windows Run on an Altair 8800?

- There is clearly a very strong correlation between software and the capabilities of the underlying platform.

- **Platform independence must not mean platform ignorance**
  - Particularly for real-time/embedded systems
A Short Digression on Terminology

- “Non-functional” (vs “functional”) requirements?
  - The term does not tell us what something is but what it is not
  - Implies and is typically interpreted as having secondary significance
  - Consequence: “Non-functional” concerns are typically addressed after “functional” ones have been resolved
    - this is actively encouraged and taught as a good approach to software design

- Preferred terminology: “Quality of Service (QoS)” e.g.,:
  - QoS requirements
  - QoS-constrained systems
Platform: the full complement of software and hardware required for an application program to execute correctly.

NB: Software engineering is very weak on methods for specifying platform requirements of software applications.

A platform also acts as a gateway to the physical world.
Platforms as Service Providers

The relationship between applications and platforms can be represented as an instance of the client-server pattern:
- NB: Most platforms can support multiple independent applications.
- Services are often shared by multiple applications.

To deal with hardware platforms, we must generalize the concept of service to include more than just software API-type services:
- CPU (processing) service
- Special device services (e.g., sensors and actuators)
- Storage service, etc.
We need our platforms to provide the necessary QoS to ensure the correct operation of our software.

The capacity (QoS) of a platform to support a given application is fundamentally constrained by the physical limitations of the underlying hardware:

- Memory capacity and latency
- CPU speed
- Communications bandwidth and latency
- Reliability and availability
- ...etc.
Quality of Service:

- **Quality of Service:** the degree of effectiveness in the provision of a service
  - e.g. throughput, capacity, response time

- The two sides of QoS:
  - *offered QoS*: the QoS that is available (supply side)
  - *required QoS*: the QoS that is required (demand side)
Key analysis question: Does a service (platform) have the capacity to support its clients?

- i.e., does supply meet demand?

Key question: \((\text{RequiredQoS} \leq \text{OfferedQoS})\) ?
Co-located applications are often designed independently

- It can be extremely difficult to predict how they will affect each other
- Fortunately, a number of methods have emerged for certain key types of QoS (e.g., schedulability analysis, queueing theory)
Platforms and Components

- The ability of a component to meet provided QoS obligations to its peers (clients) is fundamentally dependent on the QoS it requires of the platform.

- For a component to work successfully, it must be deployed on a platform capable of meeting its QoS requirements...which is why we must care about SAPs.
Platforms and Service Provision Points

- To help us match components with platforms, it is useful to introduce the *service provision point* (SSP) concept
  - The interfaces (ports) through which a platform provides its services
  - Convenient point to attach QoS information
What About Platform Independence?

- Components with SAPs and required QoS specs are **platform independent**, since the required platform is only specified in terms of QoS rather than a specific implementation technology.

- Platform independence does not mean platform ignorance, but an explicit technology neutral specification of acceptable platform QoS characteristics.
Components are often configured to co-exist within an environment with specific QoS characteristics

- E.g., security environment, operating system, communications network, component container

**QoS Domain:**

* A virtual platform such that all SPPs of a given interface type have the same QoS characteristics
Acceptable Platforms

- A platform specification (based on the canonical platform model) that identifies the range of possible platforms that are suitable for a given component-based application
  - Consists of service instances, SPPs, and (optionally) QoS domains
The Acceptable Platform Pattern

- Specifies the application “deployed” on its corresponding acceptable platform specification
  - A platform independent specification that can be mapped to any platform that conforms to the constraints specified by the acceptable platform
An application can be deployed to any platform that satisfies the constraints of its acceptable platform. ⇒ platform independent application
Overview

- The 3D Component Model
- On Software Platforms
  - The MARTE UML Profile Standard
- The CERAS MDD Research Initiative
Architecture of the MARTE specification

Foundations for RTE systems modeling and analysis (e.g., time model, resource model)

Specialization for precise modeling of RTE systems (e.g., CPUs, concurrency threads)

Specialization for formal analyses of RTE systems (schedulability, performance)

Slide courtesy of Sebastien Gerard, CEA-LETI
VSL: The Value Specification Language

- Language to specify non-functional (QoS) property values
  - Textual language
  - Includes literals, variables and expressions
  - Expressions involving variables can capture functional relationships between values of different properties

- Examples:
  - \([1..5]\) = interval literal
  - \(\{1, 2, 4, 8\}\) = numerical collection literal
  - 2008/01/31 Thr = date literal
  - (2, us) = tuple literal (for structured data) or (value=2, unit=us)
  - in $temp : Temperature = 0 = a variable declaration
  - ((temp>=0) ? 'positive' : 'negative') = conditional expression
  - aComplexNum.real = reference to “real” property of aComplexNum
• MARTE defines a model library with predefined QoS types that can be used to define custom QoS types

• Example:
  - Defining a new frequency type (MyFreqType)

```
<<nfpType>>
NFP_Real
value : Real

<<enumeration>>
FrequencyUnitKind
<<unit>> Hz
<<unit>> KHz
<<unit>> MHz
<<unit>> GHz

<<nfpType>>
MyFreqType
unit : FrequencyUnitKind

Oscillator
rate : MyFreqType = (value = 10, unit = KHz)
```
Example: A hardware platform with specified QoS parameter values

```
«hwResource»
ProcessingNode

«hwProcessor»
  : CPU
  {isSynchronous = true
   mips = 5,
   nbCores = 2}

«hwBus»
  : Bus
  {isSynchronous = true}

«hwRAM»
  : RAM
  {isSynchronous = true
   isStatic = false}

«hwDMA»
  : DMA
  {nbChannels = 2}

«hwDrive»
  : Disk[2]
  {memorySize = (300, GB),
   timing[1] = (, averageAxTime, (5, ms)),
   timing[2] = (, maximumAxTime, (50, ms))
```
There is a common form to many kinds of quantitative analyses

Demand Side

Work demand arrivals (Workload Generator)
(e.g., event arrivals, time triggers)

Work Characterization
(e.g., application programs, system programs, etc.)

Supply Side

Resource1
(e.g., disk)

...

ResourceN
(e.g., CPU)
Objective: Automating Complex KPI Analyses

- Reduces need for rare and expensive analysis expertise
- Ensures validity of results
Overview

- The 3D Component Model
- On Software Platforms
- The MARTE UML Profile Standard
- The CERAS MDD Research Initiative
Centre of Excellence for Research in Adaptive Systems

- Supported by IBM Canada (Center for Advanced Studies (CAS)) and the Ontario Centres of Excellence (CCIT)
- Other industrial partners will be joining
- 3-year mandate
- Funding: $2.7M Cash + $1.7M in-kind
  - To be leveraged further with NSERC funds, etc.

Two principal and complementary themes:

- Virtualization technologies
- Model-driven development

Web page:

- https://www.cs.uwaterloo.ca/twiki/view/CERAS/CerasOverview
Technical Objectives of CERAS MDD Thrust

1. Define a systematic and comprehensive conceptual framework (map) for MDD
   - Serves as a foundation for current and future research and development
   - Captures a shared consensus of the technical objectives behind MDD

2. Initiate a number of major research efforts to fill in crucial parts of the MDD research map

3. Act as a brokerage for MDD research worldwide
Research: CERAS Research Areas

- Specifying Models of Software
- Model Transformations
- Model Analysis and Verification
- Model-Driven Development Methods
- MDD Tooling
How Should It be Done?

- **Synergistic**
  - Maximize cross-fertilization between projects
  - Work within a common conceptual framework
  - Shared industrial partners/problems (but different aspects)
  - Evaluate proposals and projects based on adherence to above principles

- **Relevant**
  - Seek industrial partners with concrete research issues (ideally, multi-faceted)
  - Limited number of industrial partners

- **Open/Community based**
  - Engage and collaborate with international world-class experts, researchers, and institutions
  - CERAS proceedings journal (semi-annually?)
  - Open workshops at key conferences
Summary

- For QoS-constrained software systems, the classical component model needs to be extended further to support QoS and platform interfaces.
- Platforms act as (a) the construction material of software and (b) the gateway to the rest of the world.
  - We must account for its effects in systems where those effects can be significant.
- The MARTE profile provides a convenient domain-specific language for dealing with platform effects for a class of MDE approaches based on UML.
- These and all other aspects of MDE need to be placed on a systematic scientific foundation.
THANK YOU!
-
QUESTIONS?