Tutorial

F1

Evaluating Dependability Attributes of Component-Based Specifications

Ivica Crnkovic and Lars Grunske

Day: Sunday 20 May 2007, Full Day Tutorial

Venue: Ramsey
Evaluating Dependability Attributes of Component-Based Specifications

Ivica Crnkovic, Mälardalen University
Department of Computer Science and Electronics
Box 883, 721 23 Västerås, Sweden
http://www.idt.mdh.se/~icc, ivica.crnkovic@mdh.se

Lars Grunske, School of ITEE
ARC Centre for Complex Systems, University of Queensland
4072 Brisbane (St.Lucia), Australia
http://www.itee.uq.edu.au/~grunske/, grunske@itee.uq.edu.au

Dependable Systems
Outline of the Tutorial

1. Introduction
2. Basic concepts of dependable component-based systems and dependability
3. Overview of Component Models
4. Specification and composability of dependability properties
5. Overview of the State of the Art in Component-Based Dependability Evaluation Methods
6. Session Concluding remarks

Component-based software systems

Problems of software development

- The size & complexity of software increases rapidly
- Single products become part of product families
- Software is updated after deployment
- Demands of decreasing time to market
- Costs of software development increasing

Observations of the practice of software engineering

- About 80% of software development deals with changing (adaptation, improvement) of existing software
- Time to market is an important competitive advantage:
  - Importance of incorporation of new innovations quickly
- System should be built to facilitate changes
  - Easy removal and addition of functionality
- Systems should be built to facilitate reuse
  - Easy integration of existing functions

Requirements:

- Provision of approach, technologies to facilitate
  Reuse, easy update and modification of software
**Answer: Component-based Development**

- **Idea:**
  - Separate development of components from development of systems
  - Build software systems from pre-existing components (like building cars from existing components)
  - Building components that can be reused in different applications

**Component-based Software engineering** - supporting all aspects of activities in lifecycle of components and component-based systems

**Main principles: (1) Reusability**

- Reusing components in different systems
- The desire to reuse a component poses few technical constraints.
  - Similar systems architecture
  - Good documentation (component specification…)
  - A well-organized reuse process
- …

**Main principles: (2) Substitutability**

- Alternative implementations of a component may be used.
- The system should meet its requirements irrespective of which component is used.
- Substitution principles
  - Function level
  - Non-functional level
- Added technical challenges
  - Design-time: precise definition of interfaces & specification
  - Run-time: replacement mechanism

**Main principles: (3) Extensibility**

- Comes in two flavors:
  - Extending system functionality by adding components that are part of a system
  - Extending system functionality by increasing the functionality of individual components
- Added technical challenges:
  - Design-time: extensible architecture
  - Run-time: mechanism for discovering new functionality
Main principles: (4) Composability

- Composition of components
  - \( P(c_1 \circ c_2) = P(c_1) \circ P(c_2) \)
- Composition of functions
- Composition of extra-functional properties

Many challenges
- How to reason about a system composed from components?
  - Different type of properties
  - Different principles of compositions

Compositional Reasoning

- Calculating properties of a system by combining properties of its constituents (components)

- Compositional reasoning: Function
  - If \( P(C) \) of program \( C \) is a function from input to output (pipe & filter) then the composition is modeled as a functional composition:
    - If \( S = C_1 \circ C_2 \)
      Then \( P(S) = P(C_1) \circ P(C_2) \)

Predictable assembly

- Functional composition is not always possible
- Question with extra-functional properties
  - Example: dynamic memory usage \( M \)
    - If \( S = C_1 \circ C_2 \)
      then what is the composition \( M(S) = M(C_1) \circ M(C_2) \)
  - \( M \) is not defined only by properties \( M(C_i) \), but also on properties of the platform “scheduling policy for example”
  - Information supplied with \( C_1 \) is not enough

CBSE Terminology

To make the things easier we need first some definitions...

- Software Component
- Component-based systems
- Component specification
- Component composition
- Component and sytsems properties

Predictable assembly = ability to predict properties of an assembly from properties of the involved components
Summary CBSE – basic definitions

- The basis is the Component
- Components can be assembled according to the rules specified by the component model
- Components are assembled through their interfaces
- A Component Composition is the process of assembling components to form an assembly, a larger component or an application
- Components are performing in the context of a component framework
- All parts conform to the component model
- A component technology is a concrete implementation of a component model

Software Component Definition (I)

**Szyperski** (Component Software beyond OO programming)

- A software component is
  - a unit of composition
  - with contractually specified interfaces
  - and explicit context dependencies only.
- A software component
  - can be deployed independently
  - it is subject to composition by third party.

Another definition

- A software component is a software element that
  - confirms a component model
  - can be independently deployed
  - composed without modification according to a composition standard.
- A component model defines specific interaction and composition standards.

G. Heineman, W. Council, Component-based software engineering, putting the peaces together, Addoson Wesley, 2001
Variety of component models

- The generalized definition allows different component models
  - In different domains there are different requirements and constraints
    - Different interactions (architectural styles)
    - Different extra-functional properties
    - Different integration and deployment policies

Component models classifications

- **Lifecycle.** The lifecycle dimension identifies the support provided (explicitly or implicitly) by the component model, in certain points of a lifecycle of components or component-based systems.

- **Constructs.** The constructs dimension identifies (i) the component interface used for the interaction with other components and external environment, and (ii) the means of component binding and communication.

- **Extra-Functional Properties.** The extra-functional properties dimension identifies specifications and support that includes the provision of property values and means for their composition.

- **Domains.** This dimension shows in which application and business domains component models are used.

Classifications

- **Lifecycle**
  - Modeling
  - Implementation
  - Packaging
  - Deployment

- **Constructs**
  - Interface types
  - Interface specification language
  - Interface Level (signature, contract-based, semantics)
  - Interaction

- **Domain**
  - Specific
  - General-purpose

Some of component models

- AUTOSAR
- BIP
- CCM
- Fractal
- KOALA
- EJB
- MS COM
- MS .NET
- OSGi
- PIN
- PECOS
- ROBOCOP
- RUBUS
- SaveCCM
- SOFA 2.0
  - ...

System

Client

Server

<<component>>

Identif

Input

Output

Sub 1

Sub 2

Identif

Input

Output

<<component>>
Example: Component-based embedded systems

The architectural design challenge

Distributed Software Components

Software Architecture and components

- Architecture Specification
- Structure specification
- Set of interface specification
Components and system properties

What are properties?
What are dependable systems?

Some example of properties

- Reusability, Configurability, Distributability, Availability, Confidentiality, Integrity, Maintainability, Reliability, Safety, Security, Affordability, Accessibility, Administrability, Understandability, Generality, Operability, Simplicity, Mobility, Nomadicty, Hardware independence Software, independence, Accuracy, Footprint, Responsiveness, Scalability, Schedulability, Timeliness, CPU utilization, Latency, Transaction, Throughput, Concurrency, Efficiency, Flexibility, Changeability, Evolvability, Extensibility, Modifiability, Tailorability, Upgradeability, Expandability, Consistency, Adaptability, Composability, Interoperability, Openness, Heterogenity, Integrability, Audibility, Completeness, Conciseness, Correctness, Testability, Traceability, Coherence, Analyzability, Modularity, ...


Classification of properties

- Different classification
  - Run-time properties
  - Life cycle properties
  - Run time
    - Reliability, safety, performance, robustness
  - Life cycle
    - Maintainability, portability, reusability....
- CBSE
  - Component properties
  - System properties
    - Emerging properties
Quality model in ISO 9126-I

Ivica Crnkovic, Lars Grunske: Evaluating Dependability Attributes of Component-Based Specifications

Example having source code reviews” (a Software development process quality) influences the source code in that “the number of not initialized variables” (an internal quality attribute of a software product) is minimized. This positively influences the reliability of the system (an external quality attribute of a software product).

General Concepts of the ISO/IEC 9126-1

ISO/IEC 9126-1 quality attributes

Ivica Crnkovic, Lars Grunske: Evaluating Dependability Attributes of Component-Based Specifications

Quality characteristics, sub-characteristics and attributes

Ivica Crnkovic, Lars Grunske: Evaluating Dependability Attributes of Component-Based Specifications
Other views – example: Dependability

1. Ability of a system to deliver service that can justifiably be trusted
2. Ability of a system to avoid failures that are more frequent or more severe than is acceptable to user(s)

Related to
1. Trustworthiness (assurance that a system will perform as expected)
2. Survivability (capability to fulfill its mission in a timely manner)

Dependability Challenges

- How can system quality attributes be accurately evaluated, from the specification of components properties which are determined with a certain (in)accuracy?

- Given the required system quality attributes, which properties are required from the components?

- To which extent, and under which constraints can the emerging system properties (i.e. the system properties non-existent on the component level) be derived from the component properties?

- Given a set of component properties, which system properties are predictable?

Composition of properties

What do we need to know to predict system properties from component properties?
1. **Directly composable properties.** A property of an assembly which is a function of, and only of the same property of the components involved.

2. **Architecture-related properties.** A property of an assembly which is a function of the same property of the components and/or of the software architecture.

3. **Derived (emerging) properties.** A property of an assembly which depends on several different properties of the components.

4. **Usage-dependent properties.** A property of an assembly which is determined by its usage profile.

5. **System context properties.** A property which is determined by other properties and by the state of the system environment.

---

**Example**

- “Physical characteristics”
  - Static memory
    
    \[ M(A) = \sum_{i=1}^{n} M(c_i) \]
    
    where
    
    - \( M \) = memory size
    - \( A \) = assembly
    - \( c_i \) = components

    - (the “function” can be much more complicated)
    - (the functions are determined by different factors, such as technologies)
2. Definition: An architecture-related property of an assembly is a function of the same property of the components and of the software architecture.

\[ A = \{c_i : 1 \leq i \leq n\} \]
\[ P(A) = f(P(c_1), P(c_2), \ldots, P(c_n), SA) \]
\[ SA = \text{software architecture} \]

- Consequence: System/assembly architecture must be known
  - Ok when building systems of particular class
  - (product-line architectures)

3. Definition: A derived property of an assembly is a property that depends on several different properties of the components.

\[ A = \{c_i : 1 \leq i \leq n\} \]
\[ P(A) = f \left( \begin{array}{c}
P_1(c_1), P_1(c_2), \ldots, P_1(c_n), \\
P_2(c_1), P_2(c_2), \ldots, P_2(c_n), \\
\vdots \\
P_k(c_1), P_k(c_2), \ldots, P_k(c_n) \end{array} \right) \]

- Consequence: we must know different properties and their relations (might be quite complex)

Example (J2ee or .NET distributed systems)

- Client tier
- Web server tier
- Business logic tier
- Data tier

Example

- Input ports
- A
- C1
  - wcet1
  - f1
- C2
  - wcet2
  - f2
- Output ports

end-to-end deadline is a function of different component properties, such as worst case execution time (WCET) and execution period.

\[ T_{\text{end-to-end}} = L_n(i) + \sum_{c_j \in \text{aggregated}} \left( \frac{L_n(c_j)}{c_j} \right) \cdot wcet \]
4. Definition: A Usage-dependent property of an assembly is a property which is determined by its usage profile.

\[ P(A, U_k) = f(P(c_i, U'_{i,k})) : i, k \in N \]

\( P \) = attribute for a particular usage profile
\( U_k \) = assembly usage profile
\( U'_{i,k} \) = component usage profile

Consequence: It is not enough to know which system will be built. It must be known how the system will be used.

Example: Reliability

- the probability that a system will perform its intended function during a specified period of time under stated conditions.
- Mean time between failure
- How to calculate reliability for Software System?
  - Start from a usage profile
  - Identify probability of the execution of components
  - Find out (measure) reliability of components
  - Calculate reliability of the system

Can we predict reliability using existing usage profiles?

Reuse problem:

mapping system usage profile to component usage profile

When the known (measured) properties values can be reused?

\[ U_i \subseteq U_k \Rightarrow P_{k-min}(A, U_k) \leq P_i(A, U_i) \leq P_{k-max}(A, U_k) \]

5. Definition: A System Environment Context property is a property which is determined by other properties and by context of the system environment.

\[ P_k(S, U_k, E_i) = f(P_k(c_i, U'_{i,k}), E_i) ; i, k, l \in N \]

\( U_k \) = System usage profile;
\( E_i \) = Environment context
\( S \) = System
\( U'_{i,k} \) = Component usage profile

Consequence: It is not sufficient to know the systems and their usage, it is necessary to know particular systems and the context in which they are being performed.
Example

- safety property
  - related to the potential catastrophe
  - the same property may have different degrees of safety even for the same usage profile.

Summary - Classification

1. (DIR) - Directly composable properties. A property of an assembly which is a function of, and only of the same property of the components involved.
2. (ART) - Architecture-related properties. A property of an assembly which is a function of the same property of the components and of the software architecture.
3. (EMG) - Derived (emerging) properties. A property of an assembly which depends on several different properties of the components.
4. (USG) - Usage-depended properties. A property of an assembly which is determined by its usage profile.
5. (SYS) - System context properties. A property which is determined by other properties and by the state of the system environment.

DIR – component context
DIR – Architecture (assembly) context
EMG – Architecture and other components context
USG – Use context
Sys – System (including external environment) context

Conclusion

- Most of the emerging properties are impossible (or difficult) to predict from pure composition reasoning
- Different analysis methods of the systems are applied

A General Framework for Model-Based Quality Evaluation of Component-Based Systems

Encapsulated Evaluation Models
Operational Profiles
Composition Algorithms
Analysis Algorithms
A General Framework for Model-Based Quality Evaluation of CB Systems

- **Encapsulated Evaluation Models**
  - Independent from the deployment and the environment of a component
  - Similar to datasheets of electrical elements
  - Why?
    - Components are not self-contained and require external services
    - Components depend on the deployment environment
  - Examples:
    - WCET ← hardware platform
    - Reliability ← reliability of the external services
    - Performance ← frequency the environment calls services

- **Operational Profile**
  - Operational/usage profile $OP$ describes the usage of the component-based system
  - Example
    - Performance attributes depend on the number of requests per second from the system’s users
    - Reliability depends on the operational mode (continuous vs. on demand usage)

Operational Profile: Usage modeling and usage profile

- Intended to model external view of the use of the component
- Component reuse – also reuse of usage model
- Use of Markov chains (FSM + probability of transition between states)
  - Problem – for complex systems Markov chains become very large
  - Attempt to solve the complexity by introduction of State Hierarchy Model [Claes Wohlin & Per Runesson 1994]
A General Framework for Model-Based Quality Evaluation of CB Systems

- Composition Algorithm
  - Construction of a quality evaluation model for a hierarchical design specification
- Analysis Algorithm
  - “Extract” relevant measures of certain dependability attributes (e.g., hazard probabilities)
Safety Terminology (1)

- **(Accident).** An accident is an undesired event that causes loss or impairment of human life or health, material, environment or other goods.

- **(Hazard).** A hazard is a state of a system and its environment in which the occurrence of an accident only depends on factors which are not under control of the system.

Safety Terminology (2)

- **(Failure).** A failure is any behavior of a component or system, which deviates from the specified behavior, although the environment conditions do not violate their specification.
  - tl timing failure of a service (expected event or service is delivered after the defined deadline has expired - reaction too late)
  - te timing failure of a service (event or service is delivered before it was expected - reaction too early)
  - v incorrect result of requested service (wrong data or service result - value)
  - c accomplish an unexpected service (unexpected event or service - commission)
  - o unavailable service (no event or service is delivered when it is expected - omission)
- **(Fault).** A fault is a state or constitution of a component that deviates from the specification and that can potentially lead to a failure.

Safety Terminology (3)

- **(Risk).** Risk is the severity combined with the probability of a hazard.
- **(Acceptable Risk).** Acceptable risk is the level of risk that has deliberately been defined to be supportable by the society, usually based on an agreed acceptance criterion
  - ALARP
  - MEM
  - GAMAB

- **(Safety).** Safety is freedom from unacceptable risks
- **(Safety Requirements).** A safety requirement is a (more or less formal) description of a hazard combined with the tolerable probability of this hazard.
  - Hazard Spec. +THP/THR

Failure Propagation and Transformation Notation (FPTN)

- Failure Propagation and Transformation Notation (FPTN)
  - Introduced by Fenelon, McDermid, Nicholson, Pumfrey
- Benefits
  - Failure categorization (reaction too late(tl), reaction too early(te), value failure(v), commission(c) and omission(o))
  - First modular safety evaluation model
- Weaknesses
  - No process support
  - No tool support
  - Event-based
Failure Propagation and Transformation Notation (FPTN) Example

- Steam Boiler Example

<table>
<thead>
<tr>
<th>Valve</th>
<th>SIL=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propagation</td>
<td>Transformation</td>
</tr>
<tr>
<td>Open = Command:o</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Controller</th>
<th>SIL=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>Sensor</td>
</tr>
<tr>
<td>Intern1 Generated by [Hardware Defect] with [Probability=0.1];</td>
<td></td>
</tr>
</tbody>
</table>

CFT Example

- Component Fault Trees (CFTs)

- Component Fault Trees (CFTs)
  - Introduced by Kaiser, Grunske

- Benefits
  - First modular fault tree model
  - Failure categorization (reaction too late(tl), reaction too early(te), value failure(v), commission(c) and omission(o))
  - Tool support UWG

Analysis of the Top-Level CFT: The UWG3 Tool
State Event Fault Trees (SEFT)

- **State Event Fault Trees (SEFT)**
  - Introduced by Kaiser, Gramlich, Grunske, Papadopoulos
  - Benefits
    - Automatic generation of system-level SEFT
    - State-event based semantic
    - Tool support (www.essarel.de)
  - Weaknesses
    - Complex Evaluation
    - For real world application only simulation-based results achievable

State Event Fault Trees - Syntax

- **Semantics (transformational)**
  - Deterministic and Stochastic Petri Nets (DSPNs)
  - Used also for probability evaluation
- **Tool Support**
  - ESSaRel (Embedded Systems Safety and Reliability Analyser) Project [www.essarel.de](http://www.essarel.de)
  - Translation to DSPNs

State Event Fault Trees

- **Example (1)**
  - Fire alarm system
    - Controller unit (hardware + software), smoke sensor, sprinkler, watchdog

State Event Fault Trees

- **Example (1)**
  - Fire alarm system
    - Controller unit (hardware + software), smoke sensor, sprinkler, watchdog

State Event Fault Trees - Syntax

- **Basic Entities**
  - Component
  - State
  - Event
- **Relations and Propositions**
  - Gate (junction of causal chains – not restricted to binary or Boolean operators)
  - Temporal Order (Predecessor/Successor Relation)
  - Causal Order (Trigger-Relation)
  - Ports (State Input / Event Input / State Output / Event Output)
### State Event Fault Trees Example (2)

- **Hazard Description**
  - Fire breaks out and the sprinkler is not turned on within 10s

- **Delay**
  - $t = 10s$

- **Upon**
  - $= 1$

- **Sprinkler**

- **Sensor**

- **Controller SW**

- **Environment**

- **Controller HW**

- **Watchdog**

### HiPHOPS

- **Tabular Failure Annotations and HIP-HOPS**
  - (Hierarchically Performed Hazard Origin and Propagation Studies)
  - Introduced by Papadopoulos and McDermid in cooperation with Daimler Chrysler
  - **Benefits**
    - Automatic generation of system-level fault trees
    - Automatic generation of FMEA tables
    - Tool support/ Matlab Simulink
  - **Weaknesses**
    - Tabular failure annotations
    - Event-based

---

**HIPHOPS Example (1)**

HIPHOPS Example (2)


HIPHOPS Example (3)

- Generation of traditional fault trees
- Fault Tree+


Safety Evaluation Techniques & Generic Framework

<table>
<thead>
<tr>
<th>Method &amp; Reference</th>
<th>Encompassed Evaluation Model</th>
<th>Operational Profile</th>
<th>Composition Algorithm</th>
<th>Evaluation Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finalion, McDermid, Nichols, Prindley [13,14]</td>
<td>FPTN modules that describe the propagation and transformation of failure for one component</td>
<td>Not considered</td>
<td>Hierarchy composition of the FPTN modules + wiring input and output failure ports</td>
<td>Determination of the probabilities of the top level failure modes, summed, not tool supported</td>
</tr>
<tr>
<td>Papadopoulos, McDermid, Heiner, Smis [16]</td>
<td>Table failure annotations of (Markov) models, components, extraction of FPTN modules</td>
<td>Not considered, but possible handling of input failures generated by the environment</td>
<td>Automatic generation of system-level fault trees and FMEA-tables</td>
<td>Minimal cut sets analysis and determination of the probabilities of the top level fault tree nodes, automatic, with commercial fault tree tools</td>
</tr>
<tr>
<td>Grunke, Kranich, Engeler, Michel [11,10,12]</td>
<td>Component Fault Trees (CFTs), stochastic and deterministic, and decomposable fault trees where the interfaces are described by typical input and output failure ports</td>
<td>Not considered, but possible handling of input failures generated by the environment</td>
<td>Model-based construction of hierarchical CFTs based on the architecture of the system (writing input and output failure ports based on the system’s failure flow)</td>
<td>Determination of k-out-of-n probabilities (CFT + k connected, k independent or dependent, or k connected with k independent and k dependent) + model-based simulation and risk assessment with BDD algorithms (e.g. WOSA, <a href="http://www.ewid.com">www.ewid.com</a>)</td>
</tr>
<tr>
<td>Kranich, Grunke, Papadopoulos [15,16]</td>
<td>State/event-based fault trees (SEFT), stochastic based on stochastic Petri nets</td>
<td>Not considered, but possible handling of input failures generated by the environment</td>
<td>Model-based construction of hierarchical SEFTs based on the architecture of the system, based on failure propagation and fault hiding</td>
<td>Determination of k-out-of-n probabilities (SEFT + k connected, k independent or dependent, or k connected with k independent and k dependent) by simulation of a stochastic Petri Net (e.g. EESim, <a href="http://www.ewid.com">www.ewid.com</a>)</td>
</tr>
</tbody>
</table>

Safety Evaluation Case Study

Safety Evaluation of a Computer Assisted Braking System with SaveCCM

SaveCCM

- SaveCCM is a architecture description language for embedded control applications in automotive (vehicular) systems.

SaveCCM Syntax

- SaveCCM benefits for safety eval.
  - Strongly Encapsulated Interfaces
  - Hierarchical (De)Composition

SaveCCM vs. FPM

Failure Modes of Components

- Assumption
  - Components exchange information (services, messages, etc.) only via ports
- Derivation from expected information is called a failure
- For each service / message that some component produces or consumes, different failure modes can be assigned, e.g.
  - Value failure
  - Timing failure (too early / too late)
  - Omission failure (service not delivered when requested)
  - Commission failure (undesired service provided)
- As ports in structural models designate information / service propagation (failure propagation)
Safety Evaluation Process

Safety Evaluation Steps
1. Generate an encapsulated failure propagation model for each SaveCCM Component and Switch.
2. Identify the relations between system output failures and hazards.
3. Construct an encapsulated failure propagation model for each SaveCCM Assembly.
4. Calculate the output failure probabilities of the system-level Assembly and accordingly the hazard probabilities of the system.
5. Compare the calculated hazard probabilities with the tolerable hazard probabilities.

Case Study

Computer Assisted Braking System

Step 1

Step 2

Repeat this for all components
Step 3-5

Locational relationship between system-level output failures and hazards

FPM for all components

Safety Evaluation Exercise

Industrial Metal Press

Industrial Press: operational concept

Industrial Press: system-level view

Press main functions:
- Raise plunger to top (open the press)
- Release plunger (close the press)
- Abort operation (stop closing & reopen the press)

System-level requirements/operational concept:
- Upon start-up, press will open fully
- If button is pushed while press is fully open, press will start to close
- Upon closing, press will automatically reopen
- If safe to do so, closing can be aborted by releasing the button
  - Safe = above Point of No Return (PoNR)
Industrial Press: Press physical architecture

Questions

- What are the safety requirements?
- What are the system hazards?
- What are the tolerable hazard rates?
- What are the relations between system failures and system hazards?
- What are the encapsulated evaluation models?
- What are the component failure probabilities?
- Is the system safe?
- How could the system be improved?

Industrial Press: Control Logic

Answers

Note: this is not necessarily a good design
Open Problems and Future Work

- How can we determine the probability of an internal software defect or fault?
  - Empirical data
  - Measurement-based models
  - It is hard to determine the resulting failure modes for a given fault
- Effort for the COTS component vendors to produce the failure propagation models
  - All stakeholders must use compatible models / failure categories
  - Reuse potential promises pay-off

Performance, Realtime

Real-time systems

RT Systems: Correct result at the right time

Example:

Collision

Too early

Too late

An air bag must not be inflated too late, nor too early!

What are Real-Time Systems

- A Real-Time System has a number of Tasks (Programs)
- A Task has a set of properties
  - Deadline (D)
  - Period / Minimum interarrival time (T / MINT)
  - Worst/Best Case Execution Time (WCET / BCET)
  - Transactions with an end-to-end deadline (E2ED)
- A real-time system has a scheduler that uses a scheduling policy to assign a task a fraction of the processor time
Scheduling Analysis

Schedule: assignment of all jobs to available processors, produced by scheduler.
Valid schedule: All jobs meet their deadline.

Scheduling Algorithms:
- Earliest Deadline First (EDF)
- Rate Monotonic scheduling (RM)
- Deadline Monotonic scheduling (DM)

Task $T_i$ is a series of periodic jobs $J_{ij}$. Each task has the following parameters:
- $p_i$: period, minimum interrelease interval between jobs in Task $T_i$.
- $e_i$: maximum execution time for jobs in task $T_i$.

Simple classification of scheduling algorithms:

- **Online**
  - Priority based
  - Static priorities: RM, FPS
  - Dynamic priorities: RM-PIP, EDF

- **Offline**
  - Time triggered

Offline scheduling:
**Also known as static or pre-run-time scheduling**
Static schedule (time table) created before we start the system.
Run-time dispatching: just follows the generated time table.

Properties (compared to online scheduling):
- (+) Allows more complex task models
- (+) More difficult scheduling problems
- (-) Less flexible

Analysis:
“proof by construction”
Online vs offline scheduling

Online scheduling
(+): flexible
(+): relatively simple analysis
(-): difficult to cope with complex constraints
(-): less deterministic

Offline scheduling
(+): deterministic
(+): simpler to test and verify
(+): handles complex constraints
(-): new schedule must be generated if we add a new function
(-): it could take a long time to produce a schedule

When to use each of the methods?

Offline scheduling
- High demands on timing and functional verification, testability and determinism
- Safety-critical applications, e.g., control system for Boeing 777

Online scheduling
- Demands on flexibility, many non-periodic activities
- Example: multimedia applications, webservers,

Combination of both
- Combined offline and online scheduling
- The time critical parts scheduled offline and non-critical parts online

From component model to RT execution model

Design-Time:
- Component model
- Real-Time Analysis
- Synthesis
- Model transformation
- Target application

Compile-Time:
- Attribute Assignment
- Glue Code Generation
- C-compiler

Run-Time:
- Component code loading
- Target Compiler
- RTXC
- Target application
Allocating components to real-time tasks

- Today one-to-one allocation is commonly used
  - Not efficient in terms of cpu-overhead and stack usage
  - However, highly analyzable
- How can the mapping between components and tasks be analyzable and efficient?
  - Infeasible to calculate due to the many different possible mappings in a large system
- Limitations
  - Only pipe-and-filter architectures
  - No advanced real-time constraints

Parametric Contracts

- Lifting the Design-by-Contract Principle to Software Components
- Linking the provided and required services of the same component
- Specified by the QML+ Service Effect Automata

Stochastic Petri Nets

- Petri Nets
  - Places, Transition, Token
- Petri nets are extended by associating time with the firing of transitions, resulting in timed Petri nets.
  - A special case of timed Petri nets are stochastic Petri nets (SPN) where the firing times are determined by random variables.
  - Exponentially distributed firing times
  - Generalized SPN (GSPN)
    - Transition with zero firing times

Performance Evaluation Techniques & Generic Framework

<table>
<thead>
<tr>
<th>Method &amp; Reference</th>
<th>Encapsulated Evaluation Model</th>
<th>Operational Profile</th>
<th>Composition Algorithm</th>
<th>Evaluation Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wu, McAllister, Woodside [24,25]</td>
<td>Layered queuing networks (LQN), which provide a hierarchical black-box view of the performance of a component</td>
<td>Modeled as input queue</td>
<td>Automatic generation of a layered performance model from component auto-modules, tool supported</td>
<td>Traditional evaluation of the top-level LQN</td>
</tr>
<tr>
<td>Furu, Becker, Remander [29]</td>
<td>Parametric performance contracts, similar approach as the reliability evaluation described in [21]</td>
<td>Service effect automata, similar to [21]</td>
<td>Hierarchical composition of the parametric contracts and service effect automata</td>
<td>Calculation of the time consumption of possible call sequences including forks and choices, not explained in detail</td>
</tr>
<tr>
<td>Bartolino, Mirandola [52]</td>
<td>Performance optimization confined to QML's SPT profile [97]</td>
<td>Weighted use case with call probabilities</td>
<td>Construction of a formal model (queuing network) based on the performance annotations, deployment architecture and resource usage, supported by XAM transformations</td>
<td>Response times are calculated with standard queuing network analysis algorithms and tools</td>
</tr>
</tbody>
</table>
Very simple model for terminating batch sequel systems [late 70ies]

- $Comp$ is the set of components that can be called.
- $q_i$ is the probability that the component $C_i$ will be called and $r_i$ is the binary reliability of the component $C_i$ (ether the component will produce the correct output or not).
- The reliability of the system can be determined as follows:

$$R = \sum_{C_i \in Comp} q_i r_i$$

- The problems of this model are obvious

User Oriented Software Reliability Model [Cheung 80]

- Assumptions:
  - The operation profile of the system is defined by the probabilities of the transfer of control between component
  - This control transfer follows Markov-properties
  - System has exactly one start and one end-component

- Notation

  - $R_i$ reliability of component $N_i$
  - $P_{ij}$ probability of correct control transfer from component $N_i$ to component $N_j$

$$Q = \begin{bmatrix}
N_1 & N_2 & \cdots & N_{N-1} & N_N \\
0 & R_1P_{12} & \cdots & R_{1,N-1} & R_1P_{1N} \\
\vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & \cdots & 0 & 0 \\
N_{N-1} & 0 & \cdots & 0 & 0 \\
N_N & 0 & \cdots & 0 & 0 \\
\end{bmatrix}$$
User Oriented Software Reliability Model [Cheung 80]

- Probability calculation trick: Let $S$ be an $n$ by $n$ matrix such that
  $$ S = I + Q + Q^2 + Q^3 + \cdots + \sum_{k=0}^{n-1} Q^k. $$
  $$ W = I - Q $$

- [Cheung 80] shows that $S = W^{-1} = (I - Q)^{-1}$ and as a result the reliability of a system can be calculated as follows:
  $$ R = S(1,n)R_n $$

User Oriented Software Reliability Model [Cheung 80]

- $P^n$ is the the $n$th power matrix of $P$
- Consequently, $P^n(i,j)$ is the probability of reaching state $N_j$ from the starting state $N_i$ within $n$ steps
- Reliability of the system $R = P^n(N_j, C)$

Architecture-Based Software Reliability Model (1) [Wang et al. 99]

- Based on the [Cheung 80] model
- Extension
  - Multiple entry points & multiple exit point
  - Realistic operational profile
- Extension for Architectural Styles
Architecture-Based Software Reliability Model (2) [Wang et al. 99]

- Batch-sequential/pipeline style
- Analysis: identical to [Cheung 80]
- Parallel/pipe-filter style
- Analysis

Architecture-Based Software Reliability Model (3) [Wang et al. 99]

- Fault Tolerance
  - Primary component C2 and a set of backup components
  - Analysis: Reliability (by Induction)
    \[
    R_S = \sum_{j=1}^{k} \left( \prod_{i=1}^{j-1} (1 - R_{C_i}) \right) R_{C_j}
    \]
    times transition probability
  - Assumption: Independent Failure
- Call-Return
  - Analysis: identical to [Cheung 80]
  - Problem: Loop

Architecture-Based Software Reliability Model [Wang et al. 99] Example

\[ S_1, S_2, S_3, S_4, S_5, S_6, S_7, S_8, S_9, S_{10} \]

Architecture-Based Software Reliability Model with Error-Propagation [Cortellessa, Grassi 07]

- Based on [Cheung 80] and [Wang et al. 99]
- Each component has two reliability metrics
  - Internal failure probability \( inf() \)
  - Error propagation probability \( ep() \)
Architecture-Based Software Reliability Model with Error-Propagation [Cortellessa, Grassi 07] Example

- Results are more realistic
- Component Selection is more accurate

<table>
<thead>
<tr>
<th>$C_i$</th>
<th>$C_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
<th>$C_5$</th>
<th>$C_6$</th>
<th>$C_7$</th>
<th>$C_8$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Q$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$e$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\text{int} f(i)$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sensitivity Analysis

- Find the component $C_i$ with the most influence on the system reliability
- Identical to identifying architecture optimisation points, like:
  - Bottleneck (Performance)
  - Single Point of Failure (Safety)

- With respect to the component reliability [Cheung 80], [Wang et al. 99], [Cortellessa, Grassi 07]: $\frac{\partial R_{\text{rel}}}{\partial \text{int} f(i)}$

- With respect to the error propagation probability [Cortellessa, Grassi 07]: $\frac{\partial R_{\text{rel}}}{\partial \text{ep}(i)}$

Reliability Evaluation for Service Oriented Architectures

- Based on the [Kubat 89] model (formulation in the SOA domain is still pending)
- Notation:
  - $K$ describes a set of services of a system
  - $r_k$ is the service call arrival rate (Operational Profile)
- Solution:
  $$\lambda_5 = \sum_{k=1}^{K} r_k [1 - R(k)].$$
  - $R(k)$ is calculated traditionally based on the number of visits for each component and the component reliabilities when the task is called.
  - The architecture is a DTMC with transition probabilities $p_{ij}$ between components

Sensitivity Analysis – Example

Source [Cortellessa, Grassi 07]

Sensitive Component $C_2$, $C_4$, $C_7$, $C_8$
Further Models and Readings

- Classification of [Goseva-Popstojanova, Trivedi 01]
- State based models
  - Reliability Prediction and Sensitivity Analysis Based on Software Architecture [Gokhale et al. 02] [Gokhale, Trivedy 98]
- Software Dependability [Kanoun, Sabourin 87]
- Laprie model for dynamic failure behavior [Laprie84] [Laprie, Kanoun 92]
- Littlewood model [Littlewood 1979]
- Path based model (eg. [Yacoub et al. 99])
- Additive models (eg. [Xie, Wohlin 95])

Open Problems and Future Work

- How can we determine the probability of an internal software defect or fault?
  - Empirical data
  - Measurement-based models
- It is hard to determine the resulting failure modes for a given fault
- How can we determine the transition probabilities
- What are the limitations and assumptions of these models

SHARPE: Symbolic-Hierarchical Automated Reliability and Performance Evaluator

- Robin A. Sahner & Kishor S. Trivedi
- Evaluation Backend for multiple input models

Probabilistic Model Checking

- Probabilistic model checking question:
  - What's the probability of reaching bad state?
- Model
  - CTMC, DTMC, GSPN, ...
- Property Specification
  - CSL (Continuous Stochastic Logic)
  - PCTL (Probabilistic Computation Tree Logic)
- Model Checker
  - PRISM
  - ETMCC
  - VESTA
- Problems: State Explosion, Limited Support of Counter Examples
Application of Dependability Evaluation Techniques

Dependability Optimisation

How Can Quantitative Architecture Evaluation be USED in practices

Background Dependability Optimisation: Simple Solution

- Goal: Quality improvement by architecture transformation
- Solution:
  - Evaluation algorithms to determine the quality of the architecture (eg. Component Fault Trees (CFTs) → safety)
  - Transformation operators:
    - Improve the non-functional properties
    - Preserve the functional properties
  - Search with Backtracking

Architecture Transformation: Quality improving transformation operators

- Two-Channel-Redundancy
- Recovery Block
- Hardware Platform Reassignment
- Process Fusion
- Further transformation operators /Viking-Plop 2003/
  - Multi-Channel-Redundancy with Voting
  - Protected-Single-Channel
  - Hardware Platform Substitution
  - Hardware Platform Reassignment
  - Actuation-Monitor
  - Integrity Check
  - Watchdog
All Problems Solved???

- How to improve dependability aspects early in the system development lifecycle?
  - Rigorous assessment, evaluation and analysis of design specifications (architecture specifications)
    - because the earlier a quality problem can be identified, the better and more cost effectively this problem can be fixed.
  - Dependability Improving Action → Early in the development process
  - Problem: Dependability requirements conflicting with each other.
    - Trade-Offs

- Motivation
  - The fulfilment of dependability requirements is very important for the success of a software project.

---

Trade-off Analysis Method

TAFES Framework (Trade-off Analysis For Embedded Systems)

---

General Introduction to Multiobjective Optimisation Problems

- Multiobjective Optimization Problem
  - Find a solution $x$ which is an element of the solution space $X$, satisfies a set of constrains $g(x)$ and optimizes a vector function $f(x) = \{f_1(x), f_2(x), f_3(x), \ldots, f_n(x)\}$ whose elements represent the objective functions.

- Pareto Optimal Solutions
  - Set of non-dominated solutions
    - a solution $x_1$ is dominated by another solution $x_2$ if $x_2$ matches or exceeds $x_1$ in all objectives.

---

Multiobjective Optimisation Problem for Our Problem

- Problem Definition:
  - Find a solution $x$ (an architecture design) which is an element of the solution space $X$ (set of all possible design solutions), satisfies a set of constrains $g(x)$ (economic and engineering constrains) and optimizes a vector function $f(x) = \{f_1(x), f_2(x), f_3(x), \ldots, f_n(x)\}$ whose elements represent the objective functions (fulfillment of dependability requirements).
Multiobjective Optimization and Architecture Trade-Off Analysis

- Simple Solution
  - Evolutionary Algorithms
    - Mutation operators → Architecture refactorings
    - Ranking procedure → Quantitative architecture evaluations

Example (Multiobjective Optimization)

- DLR’s BIRD (Bi-spectral InfraRed Detector)
  - Two critical functions
    - Function 1: Attitude Control Function (ACF) intended to control the satellite’s position and rotation. → needed components (1,2,3,4,5,6)
    - Function 2: Collection of infrared sensor data and the transmission of the data to the ground station. → needed components (1,2,7,8)
  - Evaluation (Cost Weight, Reliability [RBDs])

Limitation of the Approach

- Dependability Optimization for Conflicting Quality Objectives
  - Multiobjective Optimization
    - Currently based on Evolutionary Algorithms
    - Future Tasks: Tabu-Search, Memetic algorithms, Swarm-based optimisations (Particle Swarms)
    - Empirical Validation
  - General Framework for Model-Driven Quality Evaluation of Component-Based Systems
    - Safety, Performance, Reliability
    - Validation and Experiments for other Quality Attributes

Conclusion

- CBD is an attractive approach
- CBD main concern is ability of composition
- Dependability includes attributes that are either not directly composable or composable when system characteristics are known
- Instead of composability, analysis of systems are used
- CBD make the analysis easier since the analysis elements are on higher abstraction level comparing non-component based systems.
- There exits many dependability analysis – they can be applied on CB systems
References

- **General**
  - Architecture Optimisation

- **Reliability**
  - V. Cortellessa V. Grassi, A modeling approach to analyze the impact of error propagation on reliability of component-based systems, CBBSE 2007, to appear
  - J.D. Musa, Operational profiles in software reliability engineering, IEEE Software 10 (2) (1993) 14–32.

- **Performance**

- **Safety**

Ivica Crnkovic, Lars Grunske: Evaluating Dependability Attributes of Component-Based Specifications