A Framework for Managing Assumptions made by Software Components in Real-time Systems

Ajay Tirumala
tirumala@uiuc.edu
Outline

• **Background and motivation**
• Case studies
• Design Concepts
• Solution approach
• Extensions and comparison with related tools
Background: Building real-time systems

Critical real-time systems are built using
a) COTS components
b) components from diverse development groups

Simple Car Control Systems

Control Systems in satellites
COTS Adoption and Concurrent Development

Significant Benefits

- Distributed expertise
- Uniform process for building different types of systems
- Faster turn around time
- Software / Hardware Component reuse.

Problems (Real-time domain)

- Gives rise to black-box software interfaces.
  - Acoustic sensor returns a 16-bit value when data is available.
  - Event `dataReady(uint_16t data);`
- Software components make assumptions which are not reflected in the software interface
  - E.g: Max sensing delay < 10ms, sensing jitter < 1ms, acoustic intensity units = {Db}, saturation value for readings <= 1023, granularity of readings <= 0.1 Db.
Real-world example (Avionics)

- Ariane 5 reused some software developed for Ariane 4
- Ariane 4 made the following assumption
  - “The horizontal velocity component will not overflow a 16-bit variable”
- This was true for Ariane 4, but not for Ariane 5.
  - This triggered self-destruction roughly 40 seconds after the launch

Assumption was known before hand but was not recorded in a verifiable (machine-checkable) format
Real-world example (Automotive sector)

- Fatality due to a software controlled airbag deactivation.
- In presence of child seat
  - Airbag was deactivated by the primary controller.
- Under certain combination of environmental conditions
  - primary controller gives the control to a simpler backup controller
- Backup controller has simple logic
  - Deploy all airbags on impact
- On impact, deploys airbag and causes fatality
  - Unaware of the environmental factor - child-seat presence  OR
  - Has the capability of only deploying all or nothing.

Environmental assumption propagation not handled during integration of the primary and backup airbag controllers
Outline

• Background and motivation
• **Case studies**
• Design Concepts
• Solution approach
• Extensions and comparison with related tools
Case-study hypotheses

• Hypothesis I: “Significant portion (>= 50%) of software defects in released products in RT-systems are related to inconsistent assumptions”
  – This hypothesis is related to the necessity of the assumption management framework.

• Hypothesis II: “Majority of the assumptions that result in defects can be encoded in a machine checkable format”
  – Thus can warn of defects in advance or prevent mismatched assumptions in end-products.
  – This hypothesis is related to the feasibility of such a framework.
Projects selection for case studies

• **Criteria**
  – Should be representative projects in the domain in which we intend to generalize the results
    • Should have close interactions with the operating environment
  – Should provide adequate data for validating the hypothesis.

• **Projects selected**
  – TinyOS [for Hypothesis 1&2]
    • an operating system for embedded devices
    • public repository of defect list available
  – Iperf [for Hypothesis 1&2]
    • an internet end-to-end bandwidth measurement tool
    • public repository of user-problems faced [mailing list available]
  – Inverted Pendulum Control System [for Hypothesis 2]
    • A system that balances an inverted pendulum using feedback control
TinyOS

- **Description and Functionality**
  - Operating System for wireless embedded sensors
  - Designed for low overhead, small memory footprint, and low power consumption
  - Event driven architecture, with single shared stack.
  - Events and tasks/commands are 2 basic constructs for execution.

- **Assumptions:**
  - TinyOS \(\rightarrow\) hardware, operating environment and applications that run on it.
  - Applications \(\rightarrow\) TinyOS, for the interpretation of the data and commands to request the data.
Iperf project functionality

• Description and Functionality
  – Measures end-to-end achievable bandwidth between hosts in an IP based network
    • Uses TCP itself to measure the amount of data sent
    • Approximates cumulative bandwidth sent over a period of time to the achievable bandwidth
  – Measures jitter and loss for a constant bandwidth UDP stream.

• Use cases:
  – To estimate the amount of data that can be sent between high-energy sites
  – ISP to client bandwidth estimation, ISP → ISP tests.
  – Network administrators – health monitoring

• Assumptions
  – Iperf → OS, Hardware : TCP Window size, Memory, System bus characteristics, availability of a number of OS features.

Functionality presented at SC 2001
Inverted Pendulum Control System

- **Description and Functionality**
  - Feedback control system
  - Objective: Hold the pendulum upright
  - 3 software components, one plant (hardware) proxy.

- **Assumptions:**
  - Over 40 assumptions encountered
  - E.g.: Controller assumes track length, mass and length of the pendulum, frictional constants, etc.

- **No public repository for defects**
- **Detailed study conducted on software architecture, interfaces, assumptions made by software components**

Assumption made by components: ACM SIGBED July 05 Special Issue
## Sample class of assumptions made by Iperf

<table>
<thead>
<tr>
<th>Assumptions</th>
<th>Defects caused</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Window size assumptions</strong></td>
<td>Min window sizes: Precludes testing in low-end devices</td>
</tr>
<tr>
<td>Min &gt; 2048 bytes [Code ]</td>
<td></td>
</tr>
<tr>
<td>OS specific min : Linux 256 bytes, SunOS: 4.5K [Configuration time]</td>
<td>Max window sizes: Results in under-reporting of bandwidth in high-bandwidth networks.</td>
</tr>
<tr>
<td>Max : OS constraints and root/admin policies [Configuration time]</td>
<td></td>
</tr>
<tr>
<td><strong>OS non-conformance to standards</strong></td>
<td>Longer than requested duration UDP tests</td>
</tr>
<tr>
<td>Win NT - ICMP error messages [Configuration time]</td>
<td>Precludes server side testing for multiple clients</td>
</tr>
<tr>
<td>Win32 – UDP single threaded per port [Configuration time]</td>
<td></td>
</tr>
<tr>
<td><strong>Resource related assumptions</strong></td>
<td>Incorrect throughputs in case of high bandwidth networks</td>
</tr>
<tr>
<td>System bus bandwidth &gt; achievable bandwidth [Dynamic]</td>
<td>Users obtained incorrect results trying to launch 250 threads</td>
</tr>
<tr>
<td>Adequate CPU / Memory [Dynamic]</td>
<td></td>
</tr>
</tbody>
</table>

- Compromises functionality
- Service degradation
Sample class of assumptions made by TinyOS

<table>
<thead>
<tr>
<th>Assumptions</th>
<th>Defects caused</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Value range/Data interpretation errors</strong></td>
<td>16-bit IDs allowed, causes ID mismatches</td>
</tr>
<tr>
<td>• Assumption that node ID is 8-bits [Code]</td>
<td>Value can well be in the range of long</td>
</tr>
<tr>
<td>• <em>int</em> is adequate for measuring time [Code]</td>
<td><em>send()</em> Crashes with zero length data (payload)</td>
</tr>
<tr>
<td>• <em>send()</em> command should have non-zero data length [Code]</td>
<td></td>
</tr>
<tr>
<td><strong>Hardware limitations + resource related assumptions</strong></td>
<td>Silent timer malfunction (hard to detect)</td>
</tr>
<tr>
<td>Timer granularity &gt; 10ms under overload [Runtime]</td>
<td>Silent dropping of tasks on overload</td>
</tr>
<tr>
<td>Only 7 tasks can be in the queue at any time [Code]</td>
<td></td>
</tr>
<tr>
<td><strong>Optimization related errors</strong></td>
<td>Can transmit only maximum length packets, wastage of bandwidth.</td>
</tr>
<tr>
<td><em>Bcast</em> module assumes default packet length [Code]</td>
<td></td>
</tr>
</tbody>
</table>

- **Compromises functionality**
- **Service degradation**
Outline

• Background and motivation
• Case studies
• Design Concepts
• Solution approach
• Extensions and comparison with related tools
[Traditional] Software interface

• Traditional programming language interfaces (E.g. C, Java, CORBA, nesC etc)

• Used to exchange actual data

• Mostly concerns the syntax of data exchange.
  – E.g. Interface has one 16 bit unsigned integer.

Software Interface
Data acq module: event dataReady(uint16_t dataval);
Sensor: signal dataReady(200);
Property Interface

• Used to encode the assumptions and guarantees made by the software component
  – These assumptions and guarantees are not a part of the data exchanged using software interfaces

• Assumptions may or may not pertain to parameters in the software interface

  E.g:
  – Units of the intensity data (assumed to be decibel)
  – Data collector expects maximum sensing delay to be < 50 ms

Definition: Encodes sufficient information about the environment, and the assumptions made by the component that are not a part of the software interface. Further, these assumptions are encoded in a machine-checkable format.
**Classification of assumptions: Dimension I**

**Time-frame for assumption changes**
- **Static assumptions**
  - assumptions that change only when the software changes
- **System configuration assumptions.**
  - assumptions that change only when configuration of the system changes or the hardware changes
- **Dynamic assumptions.**
  - assumptions that may change along the mission or run of the system

**Necessity for this classification**
- **Cost** of checking assumptions
  - Not all assumptions need to be checked at all times
- **Manageability**
- **Enabling different tools** for different types of checks

<table>
<thead>
<tr>
<th>DATA COLLECTOR ASSUMPTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acoustic intensity data units</td>
</tr>
<tr>
<td>= [Decibels]</td>
</tr>
<tr>
<td>Maximum sensing delay</td>
</tr>
<tr>
<td>&lt; 50ms</td>
</tr>
<tr>
<td>Max error in readings</td>
</tr>
<tr>
<td>&lt;= 10%</td>
</tr>
<tr>
<td>Max sensing jitter</td>
</tr>
<tr>
<td>&lt;= 10ms</td>
</tr>
<tr>
<td>Jitter specification units</td>
</tr>
<tr>
<td>= [ms]</td>
</tr>
<tr>
<td>Saturation value for readings</td>
</tr>
<tr>
<td>&lt;= 100</td>
</tr>
<tr>
<td>Granularity of readings</td>
</tr>
<tr>
<td>&lt;= 0.1</td>
</tr>
<tr>
<td>Valid reading constraints</td>
</tr>
<tr>
<td>: func_valid()</td>
</tr>
</tbody>
</table>
Classification of assumptions: Dimension II

**Criticality of assumptions**
- Each component has a core-functionality
- **Critical assumptions (Class A)**
  - Violation of some assumptions cause the core component functionality to be compromised
- **Non-critical assumptions. (Class B-E)**
  - Violation of certain assumptions may cause performance degradation, core functionality holds
  - Graded by the user (Class B-E)

**Necessity for this classification**
- Certifying functionality correctness [on assumption violation]
- Can be used by dependency management tools
- Service gradations

<table>
<thead>
<tr>
<th>Acoustic intensity data units</th>
<th>= [Decibels]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum sensing delay</td>
<td>&lt; 50ms</td>
</tr>
<tr>
<td>Max error in readings</td>
<td>&lt;= 10%</td>
</tr>
<tr>
<td>Max sensing jitter</td>
<td>&lt;= 10ms</td>
</tr>
<tr>
<td>Jitter specification units</td>
<td>= [ms]</td>
</tr>
<tr>
<td>Saturation value for readings</td>
<td>&lt;= 100</td>
</tr>
<tr>
<td>Granularity of readings</td>
<td>&lt;= 0.1</td>
</tr>
<tr>
<td>Valid reading constraints</td>
<td>: func_valid()</td>
</tr>
</tbody>
</table>

**DATA COLLECTOR ASSUMPTIONS**
Composition of assumptions

- **In the acoustic subsystem**
  - Each component has a property **maximum delay**
- **The acoustic sub-system is a part of the larger sub-system**
  - E.g.: Vehicle classification system
- **The integrator for the vehicle classification system**
  - Is not interested in individual guarantees like that of data analyzer, acoustic sensor, etc.
  - Is interested in calculating the next **Max Delay** of the acoustic subsystem.

![Diagram](image-url)
Composition of assumptions - II

Data collector makes quite a few assumptions which are satisfied by the sensor
• These assumptions need not be exposed as a part of the larger sub-system.

**Dimension III: Scope of assumptions**

• **Private:** Any assumption (or guar) that has matching guarantee (or assn) and need not be exposed as a part of the larger sub-component.
  – \( \text{max\_delay\_1} \), between the acoustic sensor and data collector

• **Public:** Any assumption (or guar) that needs to be exposed as a part of the sub-component.
  – \( \text{total\_max\_delay} \): Sum of delays of components

• **Important:**
  – Public and private assumptions are only for easier manageability
  – Checks may still need to be performed for functionality and/or performance conformance on relevant changes.

---

Match all assumptions that have matching guarantees.
[Need not be exposed in the larger sub-system]

All unmatched assumptions need to be satisfied by external components
[Part of the larger sub-system’s assumptions]

Create new assumptions and guarantees, if needed
[For example, the \( \text{total\_delay} \) in acoustic system, \( \text{loop\_delay} \) in Inv Pend system]
Making assumptions explicit:
[Context and benefits]

- **Iperf makes a key assumption**
  - “TCP slowstart is < 2-3 seconds for most networks” [hence default measurement time of 10s is adequate]
  - In high-bandwidth networks, 1Gbps-200ms transfer, slow-start duration = 5.6 seconds
    - Assumption violation causes critical error
- In most networks, e.g., 10Mbps-5ms transfer, slow-start duration << 2 seconds
  - Assumption on default measurement time causes more than required data to be transferred. 1-2 second transfer is adequate. Non-critical error.
Making assumptions explicit II: [Solution]

- Make the TCP slow-start explicit (machine checkable)
  - Use Web100 – exposes few TCP kernel variables in Linux proc file system
- New algorithm:
  - Detect end of slow-start, use TCP variables from Web100
  - Measure steady state bandwidth for a short period of time after end of slow-start.
- Results
  - Estimates differing by < 10% for 20 second standard Iperf runs.
  - Estimation time savings ~ 92%
  - Network traffic savings ~ 94%
Outline

• Background and motivation
• Case studies
• Design Concepts
  • Solution approach
• Extensions and comparison with related tools
Assumption specification within

```
system Sensor
features -- ... Filter system features
annex assumptions {**
    componentAssumptions name= DataCollector {
        -- Dependent component DataCollector
        about Sensor {

            -- Assumptions
            assumes validReadings (int validReadingsID, int startLatency, int frequency) {
                validReadingsIDs > startLatency/frequency
            }
            {Criticality=CRITICAL_LEVEL_5}
            {ChangeInterval=STATIC};

            -- Guarantees
            guarantees minFrequency {
                int frequency = 100;
            }
            ;
        }
        about OtherComponents {
            ...
        }
    }
**}
```

Sensor

Data Collector

Makes assumption validReadings

Provides guarantee minFrequency
Assumption management: Objects defined using EMF

[Diagram from Peter Feiler’s presentation]

EMF - The Data Integration Foundation of Eclipse
XML persistent objects as a hierarchical database

- Example of simple search queries.
  - Check status of all *critical* assumptions.
    - //assumptions/@criticality='CriticalLevel_5'
  - Get all assumptions for a particular component ‘x’
    - //assumptionSet/@dependentComponentName='x'.

```xml
//tinyos_req/@criticality="critical"
Assn x_1
Assn x_2,...
Assn x_n

//tinyos_req/@changeInterval="system_config"
Assn y_1 ...
Assn y_n
```
Graphical editing of assumptions (EMF)
Java code generation for assumptions

- Current implementation of the parser generates Java code to check the assumptions.
  - Any assumption that can be specified with predicate logic can be checked easily.
- Can interface with tools like JavaMOP for assumptions in temporal logic
  - JavaMOP generates Java code for testing temporal logic predicates.
Composition of assumptions

- Complexity: $O(n)$ : $n$ is the number of assumptions for a complete composition.
- Can include domain specific composition rules.
- Use markers to point to the assumptions themselves.
Assumptions validity

- Checks that assumptions can be made only between compatible AADL components.
  - E.g.: *Data* component cannot make assumptions on the *Memory* component.
- Of course, flags all assumption violations.
Outline

• Background and motivation
• Case studies
• Design Concepts
• Solution approach
• Extensions and comparison with related tools
Extensions: Domain specific assumptions

- Existing dimensions
  - time-frame-of-change, criticality and scope hold for most systems.
- Domain specific case studies.
  - RT – protocol [FAI-EDF]
  - Networking applications.
- Domain specific default assumption toolkit
  - Every periodic real-time component has *period, WCET*. 
    - Depending on the system it runs, there are assumptions on priority mapping.
  - Byte-ordering (little-endian, big-endian) in networking protocols

Every component will inherit a default set of assumptions depending on the domain it is classified into.

Associated goal is to form a rich hierarchy of assumptions specific to domains.
Dynamic assumptions checking

- Challenge with RT-systems
  - How to ensure timing properties along with validation code?
  - How to handle assumption violation?

Initial studies suggest necessity for a separate VM

Dynamic Assumptions Monitoring framework

versus

RT- Application

Dynamic Assumption Monitoring framework

RT Application

Messages
Vertical assumptions tracking

- Many assumptions are generated in translating requirements to code.
  - In addition to inadequacy of software interfaces.
- Need to track assumptions across software life cycle.
  - Need a language which will help in generation of machine checkable requirements.
- Need to **revalidate assumptions** when requirements or code change.
  - E.g.: When PerfSocket_UDP.cpp change should necessitate revalidation of *UDP Buffer Size* assumption.
  - Efficiency concerns.
## Comparison with related solutions

<table>
<thead>
<tr>
<th>Design by Contract</th>
<th>- Cannot validate assumptions for assumptions not in code, management of assumptions, Composition</th>
<th>+ Can check dynamic assumptions (for assumptions that have representation in code)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT-CORBA</td>
<td>- Cannot handle [not meant] to handle most of the semantic assumptions, management of assumptions, composition</td>
<td>+ Resource related assumptions, for example thread pool management in Iperf.</td>
</tr>
<tr>
<td>VEST</td>
<td>- Cannot handle any system configuration assumptions, since it’s a modeling tool, remote from implementation.</td>
<td>+ Can model resource related dynamic assumptions</td>
</tr>
<tr>
<td>SpecTRM</td>
<td>- Design the entire project in SpecTRM from scratch. May not be suitable for COTS.</td>
<td>+ Suitable for very small critical modules, complete verification</td>
</tr>
<tr>
<td>MOP</td>
<td>- Structure of assumptions and composition. Checking assumptions not represented in code.</td>
<td>+ Check dynamic assumptions, generates code for temporal properties checking.</td>
</tr>
</tbody>
</table>
Thank you

**Integration Problem:**
Having divided to conquer, we must reunite to rule – M. Jackson

E-mail: tirumala@uiuc.edu
APX: Scalability tests

- Sample code size cell phones: > 1M lines of code.
- XML validation infrastructure: SAX or DOM parser
- Test 1:
  - Results:
    1. DOM parser is faster than SAX parser for size < 512 fragments
    2. SAX parser consistently faster than DOM
    3. DOM parser crashed with 64K fragments
    4. SAX parser successfully validated 2M fragments (48 sec)
- Test 2:
  - Nested document validation times [2-64K nesting level]
  - Results
    - DOM was better than SAX almost consistently
    - Performance difference was negligible.

SAX Parser is more scalable.
Current implementation uses SAX parser for handling events.
Do not see practical document structures with nesting levels greater than few tens or hundreds..
SAE AADL Standard
An Enabler of Predictable Model-Based Embedded System Engineering
[Slide from Peter Feiler’s presentation]

- Notation for specification of task and communication architectures of Real-time, Embedded, Fault-tolerant, Secure, Safety-critical, Software-intensive systems
- Fields of application: Avionics, Automotive, Aerospace, Autonomous systems, …
- Based on 15 Years of DARPA funded technologies
- Standard approved & published Nov 2004
- http://www.aadl.info
AADL: The Language

[Slide from Peter Feiler’s presentation]

Components with precise semantics
  – Thread, thread group, process, system, processor, device, memory, bus, data, subprogram

Completely defined interfaces & interactions
  – Data & event flow, synchronous call/return, shared access
  – End-to-End flow specifications

Real-time Task Scheduling
  – Supports different scheduling protocols incl. GRMA, EDF
  – Defines scheduling properties and execution semantics

Modal, configurable systems
  – Modes to model transition between statically known states & configurations

Component evolution & large scale development support

AADL language extensibility
System Type
[Slide from Peter Feiler’s presentation]

```system
GPS

features

  speed_data: in data port metric_speed
  {arch::miss_rate => 0.001 mps;};
  geo_db: requires data access
  real_time_geoDB;
  s_control_data: out data port
  state_control;

flows

  speed_control: flow path
  speed_data -> s_control_data

properties arch::redundancy => 2 X;

end GPS;
```
system implementation GPS.secure

subcomponents

  decoder: system PGP_decoder.basic;
  encoder: system PGP_encoder.basic;
  receiver: system GPS_receiver.basic;

connections

  c1: data port speed_data -> decoder.in;
  c2: data port decoder.out -> receiver.in;
  c3: data port receiver.out -> encoder.in;
  c4: data port encoder.out -> s_control_data;

flows

  speed_control: flow path speed_data -> c1 -> decoder.fs1
                -> c2 -> receiver.fs1 -> c3 -> decoder.fs1
                -> c4 -> s_control_data;

modes none;

properties arch::redundancy_scheme => Primary_Backup;
end GPS;