Interface Inconsistencies in Safety-Relevant Computer Systems

Francesca Saglietti
Department of Software Engineering
University of Erlangen – Nürnberg
Germany

4. Bielefeld Workshop “Systems Engineering”
Risiko-Analyse und Unfall-Ursachen-Analyse
Introduction

◆ Growing tendency to **re-use pre-developed components**
  - for **economical** reasons
  - for reasons of **reliability** (in particular for safety-relevant applications)
  operational experience gained with pre-existing components contributes to an increased confidence in the correct system performance

◆ Nonetheless
  - even proven-in-use components their integration can lead to serious software failures at system level (a. o. Ariane 5, Mars Climate Orbiter).

◆ Main **Cause**
  - components – though correctly developed – may not correctly reflect the **context** of the new system planned.
Context-Sensitive Failures

◆ **Recent accidents** prove that this is often underestimated
  - still tendency to restrict / focus V&V activities to cross-checking of specified and coded functionality at unit level
  - eventuality that implemented functions might not provide, under particular circumstances, the service required in a larger contextual domain, is rarely taken into account

◆ **Potential design gap**
  components inadvertently re-used under inaccurate assumptions may lead to
  - SW components correctly developed, but incorrectly used
  - as the functionality implemented depends on parameters assumed to fit the new application, but motivated by the original context for which they were developed
Component-Based System

Context 1

New system context

Context 2

Component 1

Component 2
Classification of Interface Inconsistencies

Solution envisaged

◆ early identification of potential inconsistencies leading to
  ◦ removal resp.
  ◦ tolerance strategies (typically by wrapping)

Procedure

◆ classify interface faults/anomalies w.r.t. different attributes, e.g.
  ◦ inconsistencies of syntactical nature
    e.g. deviation from predefined data types, signatures, etc.
  ◦ inconsistencies of semantic nature
    trespassing boundaries of physical domains

◆ define appropriate checks to identify / tolerate inconsistencies
  ◦ static checks
  ◦ dynamic checks
Classification of Interface Inconsistencies

- **Syntactical** Inconsistencies
- **Semantic** Inconsistencies
  - Logical Inconsistencies
  - Physical Inconsistencies
- **Application** Inconsistencies
  - Violation of Input Relations
  - Data Range Inconsistencies
- **Pragmatic** Inconsistencies
  - Violation of Absolute Time Constraints
  - Violation of Concurrency Constraints
  - Violation of Architectural Constraints
Syntactical Inconsistencies

- Mismatch in **data representation** at programming language level
  - lack in correspondence between representation of data imported / exported
  - example: floating point number vs. ratio of integer numbers

- **Incompatibility** can occur at different information levels
  - number of communication parameters
  - sequence of parameters invoked
  - per parameter: definition of **format** of underlying data structure including
    - size of record
    - allowed sequences of alphanumerical elements, in particular
    - usage and handling of leading zeros
    - usage and handling of blanks
    - representation of exponent and mantisse
    - Indication of decimal numbers by dot, comma, etc.
    - Indication of thousand by dot, comma, apostroph, etc.

- Standard definition of formal language
  - in general **easily** checked
Semantic Inconsistencies

- Different semantics of data exchanged between components
  - in spite of fulfilling the same syntax rules
  - deviation of semantics, possibly in different domains, e.g.

- Logical Inconsistencies
  deviation of semantics on the logical domain
  - usage of identical, syntactically correct symbols with different meaning for the underlying logic
  - e.g. by representing different numerical values

- Physical Inconsistencies
  deviation of semantics on the physical domain
  - usage of identical syntactically correct symbols with different meaning for the underlying technical process
  - e.g. by referring to different physical reference systems
Logical Inconsistencies 1

- **Same names** are used to denote different numbers
  - example: unit prefix **Mega** may denote
    - the value 1,000,000 in the metric system or
    - the value 1,048,576 used to quantify data storage

<table>
<thead>
<tr>
<th>Power of ten</th>
<th>American name</th>
<th>British name</th>
<th>Metric prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Thousand</td>
<td>Thousand</td>
<td>Kilo</td>
</tr>
<tr>
<td>6</td>
<td>Million</td>
<td>Million</td>
<td>Mega</td>
</tr>
<tr>
<td>9</td>
<td>Billion</td>
<td>Milliard</td>
<td>Giga</td>
</tr>
<tr>
<td>12</td>
<td>Trillion</td>
<td>Billion</td>
<td>Tera</td>
</tr>
<tr>
<td>15</td>
<td>Quadrillion</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>18</td>
<td>Quintillion</td>
<td>Trillion</td>
<td>---</td>
</tr>
</tbody>
</table>
Logical Inconsistencies 2

◆ Syntactically correct usage of **dots and commas** to denote different constructs

◆ Example: Mariner I Spacecraft
  (Atlas booster launch failure)
  
  ▶ FORTRAN: variables must not be declared
  ▶ led in the 70s to a loss for NASA of 3 million $ plus a satellite

◆ Both expressions
  
  ▶ DO 100 I=1.10 and
  ▶ DO 100 I=1,10

  syntactically permitted, but with different semantics
Physical Domain

- Data meant to reflect the **technical process** modelled fail to do so in a larger context than the component context.
- Therefore cannot be reused in a global context without special adaptation effort.

- Relative context to be specified by the underlying **reference system**, including:
  - Physical elements addressed
  - Physical unit
  - Currency
  - Country
  - Time zone
  - Language, etc.
Example: Mars Climate Orbiter

- **NASA, 1999: Mars Probe mission**
  Mars Climate Observer entered atmosphere

- According to Arthur Stephenson, chairman of the Failure Investigation Board
  the 'root cause' of the loss of the spacecraft
  was the failed translation of data
  representing physical forces

  - from **English** units (pound-force)
  - into **metric** units (Newton)

  in a segment of ground-based, navigation-related mission software

  - JPL controllers expected **Newton** units
  - Lockheed Martin Astronautics provided **Pound** units
Application Inconsistencies

◆ components functionality not fitting novel application context

◆ Violation of Input Relations
  constraints on component inputs may be expressed as relations between parameters and (possibly) internal states of components

  - example 1: software controlling the mixing of substances in a chemical plant. explosion caused by the usage of values indicating chemical substances whose mixture reacted in an undesired fashion.
  - example 2: for constraints on inputs and internal states consider restrictions on input parameters for components during their maintenance state.

◆ Data Range Inconsistencies
  value ranges of the same data differ for importing and exporting components

  - example: Ariane 5
**Example: Explosion in Chemical Factory**

- **Cindu (NL), 1992**
  caused the death of 3 firemen of the works fire brigade and injured 11 workers. The damage was estimated at several 10th of millions NL guilders.

- A simple typing error in a prescription by a laboratory worker led to this tragic accident.
  - Instead of tank 632 containing *resin feed classic* (UN-1268) normally used in the batch process
  - he typed tank 634 (storing *dicyclopentadiene*)

- the computerised chemical processing installation was fed with data causing the wrong amount of *chemicals to be mixed* in the installation.
Example: First Ariane 5 mission

- Explosion due to **re-use of conversion routine**
  - originally conceived for range of inputs based on flight trajectory of the predecessor version Ariane 4
  - horizontal speed exceeded predecessor
  - original conversion routine re-used to convert a 64 bit to a 16 bit number could not process data range required by new application context
    - range not sufficiently wide for new trajectory values
    - 2 identical inertial reference systems raised exception

Left: Lift off; all seemed fine for the mission. Right: Bang; flaming fragments of the Ariane 5 and its payload of scientific equipment fall to earth. Pictures: AFP
Computational Environment

Computational environment includes

- **concurrency** constraints

- **access** policies to external resources

- **timing** requirements dictated by
  - hardware and
  - user interface constraints

- underlying architectural constraints like
  - programming language paradigms
Pragmatic Inconsistencies

- **Violation of Absolute Time Constraints**
  restrictions on timing of different requests to a component; includes constraints on execution time of operations or on time between two requests
  - rare timing conditions may cause failures extremely difficult to reproduce in a controlled environment
  - example: Patriot system

- **Violation of Concurrency Constraints**
  restrictions on relative timing of component executions (explicitly specified at application level or implicitly dictated by the computational environment)
  - examples: race conditions and transactional failures in database systems

- **Violation of Architectural Constraints**
  different components operate in different architectural environments
  - example: message-based vs. object-oriented client-server environment
Example: Patriot Missile Defense

1991-1992 Operation Desert Storm
conflict between Coalition forces and Iraq, the Coalition used a military base in Dhahran, Saudi Arabia protected by 6 U.S. Patriot Missile batteries.

- Patriot system tracks and intercepts objects, such as cruise missiles or Scud ballistic missiles.
- The prediction of where the Scud will next appear is a function of the Scud's velocity and the time of the last radar detection.

Patriot battery at Dhahran failed to track / intercept Scud
- software problem in system's weapons control computer led to inaccurate tracking calculation that became worse the longer the system operated.
- At the time of the incident, the battery had been operating continuously for over 100 hours. The Patriot system was originally designed to be mobile and to operate for only a few hours in order to avoid detection.
Example: Blood Databank

COTS database management and network software

◆ 2 separate laboratories tested samples for different diseases
  ■ lab B tested for infectious hepatitis (IH), lab A tested (also) for HIV
  ■ both laboratories accessed same record simultaneously to enter test results (due to reduced blood donations, i.e. fewer blood samples to test)
  ■ lab B overwrote lab A's findings (so whether blood had HIV unknown)
  ■ freezing can destroy IH, so blood believed safe for use after processing
  ■ fortunately, company discovered the error prior distribution of the blood.

◆ Originally
  ■ database manager was designed for single computer applications without considering networked applications
  ■ could not anticipate that two users accessing a record simultaneously would lead to a hazard.
  ■ Likewise, the network program designer would rely on the database management program to provide the necessary records security.
Domains of Inconsistencies
Class-Specific Consistency Checks

- **Syntax**: static type checks of component interfaces (as standard compilers)
- **Logics**: static check of component interfaces against each other, once they have been extended to include the logical semantics of public symbols
- **Physics**: static check of component interfaces once they have been extended to include physical reference system to which public symbols refer

- **Input / State Relations**: dynamic check of component interfaces against a formalized representation of input/state relations
- **Data Range**: static check of component interfaces against a formalized representation of data ranges required by the application envisaged

- **Absolute Time**: analysis of temporal component properties against overall timing properties dictated by system computational context (preliminary static analysis to be supplemented by dynamic timing checks)
- **Concurrency**: analysis of inherent concurrency properties of components against overall concurrency properties dictated by system computational context (preliminary static analysis to be supplemented by dynamic checks)
- **Architecture**: static check of formalized descriptions concerning the component computational environments against the formalized description of the application computational environment
Classification

- Inconsistency classes, check types and entities to be checked

<table>
<thead>
<tr>
<th>Inconsistency Class</th>
<th>Check Type</th>
<th>Interface Descriptions</th>
<th>Extended Interfaces Description</th>
<th>Application Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>static</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logics</td>
<td>static</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Physics</td>
<td>static</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>State / Input</td>
<td>dynamic</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Data Range</td>
<td>static</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Absolute Time</td>
<td>static &amp; dynamic</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Concurrency</td>
<td>static &amp; dynamic</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Architecture</td>
<td>static</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Wrappers

- **Filtering** mechanisms for selecting inputs / outputs rejecting:
  - illegal data
  - untrustable scenarios
    insufficient positive experience
  - suspicious scenarios
    related to negative experience

- **Converting** mechanisms for:
  - making data compatible
Conclusions

- **Ongoing work** (by M. Jung, University of Erlangen-Nürnberg)
  - component description by extended interface description language

- For reasons of **simplicity** and **portability**
  - UML meta-model as basis for the interface definition language
  - additional information integrated by refining basic description

- **Future plans**
  - creation of component repository
  - development of component configuration tool capable of analysing repository elements in the light of application context providing automatic support for reliable assembling of component-based applications

- **For more details**
  - F. Saglietti, M. Jung
    "Classification, Analysis and Detection of Interface Inconsistencies in Safety-Relevant Component-based Systems“
    Probabilistic Safety Assessment & Management, Springer-Verlag, 2004